Università degli Studi di Milano
Facoltà di Scienze Matematiche, Fisiche e Naturali
Dipartimento di Informatica

Dottorato di Ricerca in Informatica
XXVI Ciclo
Settore Scientifico Disciplinare INF/01 Informatica

Hardware-Assisted Virtualization and its Applications to Systems Security

Tesi di
Aristide Fattori

Relatore
Prof. D. Bruschi

Coordinatore del Dottorato
Prof. E. Damiani

Anno Accademico 2012/2013

Università degli Studi di Milano
Facoltà di Scienze Matematiche, Fisiche e Naturali
Dipartimento di Informatica

Doctor of Philosophy in Computer Science
XXVI Ciclo

Hardware-Assisted Virtualization and its Applications to Systems Security

Ph.D. Candidate
Aristide Fattori

Adviser
Prof. D. Bruschi

Ph.D. Coordinator
Prof. E. Damiani

Academical Year 2012/2013

Abstract of the dissertation

Hardware-Assisted Virtualization and its Applications to Systems Security

by
Aristide Fattori

Università degli Studi di Milano

In recent years, the number and sophistication of cybercriminals attacks has risen at an alarming pace, and this is not likely to slow down in the near future. To date, security researchers and industry proposed several countermeasures to this phenomenon, and continue to investigate new techniques, in a real arms race against miscreants.

Most modern techniques to detect or prevent threats are based on *dynamic analysis*, that allows to observe the properties and behaviors of software while it runs. Many dynamic approaches are based on virtualization technology. Over the years, indeed, virtualization became the *de facto* standard environment for the implementation of many dynamic security tools and frameworks. Virtualization has many features that are particularly useful when dealing with systems security. Operating as a hypervisor (i.e., the entity that controls the execution of a system inside a virtual machine), indeed, grants a good degree of transparency and isolation, since the hypervisor is always more privileged than any component running as a guest of a virtual machine. On the contrary, approaches that directly work in the same system of their targets are prone to identification and corruption of their results.

Until some years ago, virtualization was uniquely performed via software. Due to the many challenges and intricacies of virtualization, most software hypervisors have lots of prerequisites (e.g., the source code, or binaries, of a system must be modified *before* it can be run as a guest of a virtual machine). Furthermore, they commonly have bugs, due to the enormous amount of little details that must be handled, and these badly affect transparency and isolation qualities. These pitfalls greatly hinder security systems built on top of software hypervisors. The introduction of an hardware support for virtualization on most commodity CPUs, however, provided a good mean to overcome these limitations. In a strive to contribute to the systems security research field, in this dissertation we show how such hardware support can be leveraged to build tools and frameworks that use dynamic analysis to face some of the many challenges of the field. In more details, we first describe the design and implementation of a generic framework to perform complex dynamic analyses of both user- and kernel-level software, without relying on any native support or any *a priori* modification of the target. This framework lays the foundation of this dissertation, and on top of it we built the other two contributions: a malware detector and a tool to automatically discover vulnerabilities in Mac OS X kernel modules.

To Annalisa,
for always being by my side

Acknowledgments

The first thanks goes to my external referees, Prof. Herbert Bos, Prof. Thorsten Holz, and Prof. William Robertson, for their insightful comments that improved the quality of this work.

Then, I would like to thank my advisor, Prof. Danilo Bruschi for his help and guidance during these years.

A special thanks goes to Lorenzo “gigi sullivan” Cavallaro, for being such a good friend and colleague, as well as an informal additional advisor.

Thanks to my parents and my sister, for the great support you always gave me during all these nine long years (note to the reader: including BSc and MSc!).

Thanks to my grandma, who never really got what this PhD thing was and spent three years asking me when I would stop going to school and begin to work. You played a key role in this adventure with your love.

Thanks to my colleagues, Alessandro, Srdjan, and Chiara, with whom I shared the delightful joys of these years. We’ve had our ups and downs but in the end we’ve had quite some fun spawning root shells, drinking lots of beer (ichnusa!), and re-potting plants in the lab.

Thanks to all the `LaSER` guys, it has been a privilege and a pleasure to work with you. In alphabetical order: Andrew, Brown, Cascella, DarkAngel, Ellegi, Ema, Erik, Fede, Gianz, Jack, Jeppojeps, Lollacci, Martignlo, il Nerd, Pago, Robi, Ricky, S4lv0, Stevedelik, StefanoAI, and Winnie. Thanks for the countless days and endless nights we spent playing CTFs, reversing and exploiting challenges, and hunting bugs at ring -1!

During these years I sacrificed many things, one of which was time for my old friends. Nonetheless, they’re still there for me, so thank you guys: Luca, Cucco, and Ale.

Thanks to everybody I met on the road: Miriam, Max, Vida PRO, Gabor, Davide Ariu, Fra, and many more.

Thanks to Lorena Sala, for always being so patient when dealing with me and my almost pathological inability to handle any bureaucracy and paperwork.

Finally, my most grateful thanks goes to Annalisa, who always incites me to aim at newer and farther goals, and to try to overcome my own limits. I love you, without you nothing of all I did during these years would have been possible.

For the third time, I've procrastinated this part of the thesis to the very last minute, so apologies to those that I forgot to thank :-)

Contents

1 Introduction

Systems security is of uttermost importance for our whole society. The more computer systems become pervasive in our daily lives, the more appealing they appear to criminals and terrorists. Unfortunately, the situation is not good: everyday we read reports of malware infecting millions of users, attacks against banks and infrastructures, intrusions into government facilities and so on [95, 37, 88, 113]. This phenomenon is mainly caused by the fact that actions once mainly driven by passion and by a thirst that could only be quenched by knowledge, are now motivated by an extremely different lust: money. A whole *underground economy* was born and raised on the ground of cybercrimes, and its ever-growing revenues keep pushing more and more criminals into becoming *cybercriminals*, with the promise of easy gains. Users identities, e-mail accounts, credit card numbers, shells on compromised machines, weaponized exploits: almost everything can be sold on the black market of cybercriminals [123, 53].

As if this was not enough, to face the countermeasures of security researchers and industry, miscreants are getting more and more organized. Indeed, they created complex systems to monetize the whole distribution chain of malware, automating every step to maximize the number and persistence of infections. Providers of such service then sells malware installations to their customers on the black market, in a process named *pay-per-install* by security researchers [17]. Similar services exists for exploits, rather than malware, and typically sell, or rent out, exploit toolkits [48].

According to data collected by anti-virus companies and security researchers, the cybercrime phenomenon is not going to stop its growth anytime soon. Figure 1.1 reports just one of its aspects (malware), that can however give an idea of the rate of the phenomenon growth: McAfee received almost as many malware in the second quarter of 2013 alone as in the whole 2011 [83]. Furthermore, Symantec reports that more than 5,200 new vulnerabilities were discovered and publicly disclosed in 2012 [127]—and these are probably just the tip of the iceberg, considering those that were discovered but not disclosed.

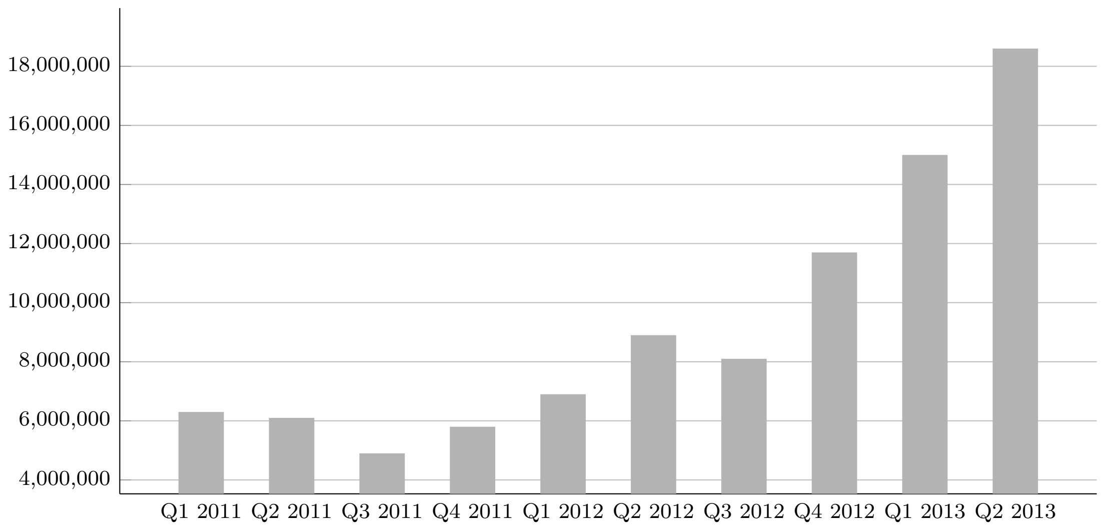


Figure 1.1: New malware samples, McAfee[1]

To fight cybercriminals and, more generally, to improve the security of programs and operating systems, much effort has been put in the creation of approaches and techniques focused on both *defense*, e.g., detection of malware and intrusions, and *prevention*, e.g., automatic identification of vulnerabilities. Such techniques are commonly based on program analysis, and are usually divided among *static* and *dynamic*, two branches that have completely separated origins, but whose boundaries are lately becoming more and more blurred, as researchers propose hybrid approaches that mix the best of both. In a glimpse—much more details will be given later in this thesis—static analysis extracts information on properties of the target and on its possible behaviors, without running it, while dynamic approaches execute the target, and observe its properties and behaviors at run-time.

Static approaches initially appeared more appealing than dynamic ones, because they theoretically allow to reason on all the properties of the analysis target, while the latter are usually less generic, as they can observe only what is exhibited at run-time. Unfortunately, this does not always hold. For example, some programs dynamically generate code and jump into it with indirect control transfers (e.g., `jmp *%eax`) that are extremely hard—or impossible, in some cases—to analyze statically. Furthermore, cybercriminals more and more commonly exploit a plethora of advanced self-defense techniques to thwart static techniques; obfuscation [72], packing [77], polymorphism [91], and metamorphism [129] are used to hinder the analysis of both malware and exploits.

In the wake of the introduction of these self-defense techniques, dynamic analysis began to be more and more used. The reason behind this is simple: indepen-

[1]McAfee report [83] does not clarify what is considered *new malware*. We assume that all variants of the same sample (i.e., with a different checksum) are considered as such.

dently from how complex or obfuscated a piece of software is, when executed it will actually do something that can be observed dynamically. Thus, by running and monitoring the target, dynamic approaches are able to understand what is hidden to the eyes of static ones. Unfortunately, dynamic is not a synonym of *bulletproof*, and special care must be taken when designing a dynamic analysis approach. Drawing a parallel between medical and security worlds, the analysis environment must be *sterile*, i.e., its presence must not affect nor corrupt the behavior of the target, and completely *isolated*, otherwise the target may, purposely or not, modify the analysis engine itself, automatically invalidating its result. These prerequisites may seem trivial to obtain, for example leveraging standard privilege separation facilities commonly offered by most operating systems, but satisfying them is everything but simple. Such requirements have been recently identified and formalized, among many others, by Rossow *et al.* [107].

Over the years, virtualization [45] became the de facto standard to fulfill the requirements of dynamic analysis. The ability to run a single program, or even a whole system, as a *guest* of a confined virtual container, and inspect its state from the *hypervisor*, i.e., the entity that controls the execution of the container, proved to be indeed appealing for the construction of many analysis frameworks and tools. Operating as a hypervisor, indeed, grants more than one benefit for an analysis engine that, theoretically, has full control over the target, can inspect its state, and can hide its presence quite easily.

Until some years ago, however, virtualization environments where still implemented uniquely via software, and analysis engines built on top of them were flawed by the limits of software virtualization. Despite they are theoretically perfect environments for dynamic analysis, indeed, software level hypervisors have more than one drawback. For example, standard software techniques require to modify the system that must be run as a guest of a virtual machine, either at the source code level, in *paravirtualization*, or at binary level, with *binary translation* [137]. Such solutions cannot be applied in every possible situation (e.g., the source code of the guest may be unavailable) and, at any rate, need to be carefully tailored around the systems (e.g., the same binary translations applied to Windows cannot be seamlessly applied to Linux). Furthermore, due to the many intricacies of their implementations, software hypervisors often have subtle bugs that can be leveraged, by software running inside the guest, to identify the presence of a virtualization environment [102]. These pitfalls clearly hinder dynamic analysis engines built on top of software hypervisors, limiting their applications to systems that can be easily run as guests of software virtualization environments, or that are not influenced by its presence, either purposely (i.e., through a red pill) or not.

The introduction of hardware support for virtualization on most commodity CPUs, however, breathed new life into virtualization-based approaches [92, 4, 46]. Without lingering on technical details, hardware-assisted virtualization allows to write flexible and transparent hypervisors, without requiring modifications of virtualized systems. Such features can be extremely useful in creating tools and frameworks to tackle many problems of systems security. Indeed, the ability to run an OS as a guest of a virtual machine, without modifying it, greatly increase the applicability of virtualization. Last but not least, transparency is much easier to obtain with hardware support. As we briefly mentioned, there are hundreds of red-pills for software virtualization environment [81], while, to date, the only detection technique for a well-configured and correctly implemented hardware-assisted hypervisors are *timing attacks* [40]. However, the applications of such attacks on real-world hypervisors on unknown hardware are still open to discussion.

1.1 Dissertation contributions

This dissertation contains details of the design and implementation of three solutions that address as many problems among those most relevant for systems security. All three solutions are strictly connected: the first prepares the ground for the other two that, as we will see, are built on top of it.

We first present the internal details of a generic framework to perform complex dynamic analyses of both user- and kernel-level software, without relying on any native support or any *a priori* modification of the target [39]. This framework is at the basis of this dissertation, as every solution that we hereby present is built on top of it. Our framework is indeed a very convenient basis for many dynamic analysis approaches, as it allows to overcome many issues of such techniques, offering a completely isolated and transparent environment and allowing to analyze *any* running target, thanks to its ability to be *hot plugged* in a system as it runs. Furthermore, it offers an API that can be leveraged to build custom analysis tools that will be executed on top of the framework. In this dissertation we also present some contributions that were not originally included in the paper, such as the support for extended page tables (EPT), the implementation of efficient and transparent EPT based breakpoints and watchpoints, as well as the addition of OS-dependent virtual machine introspection techniques for Windows 7 and Linux. Parts of the framework have also been presented in Paleari's thesis [101]. In this dissertation, we report only those parts of the framework to which we contributed directly, or that have been introduced as a novel contribution of the thesis (e.g., EPT support and all components that followed the introduction of such support).

The second contribution is a *malware detector*, built as an evolution of AccessMiner [67]. AccessMiner defines and uses *access activity models* that, with a reasonable amount of training, can be used as a basis to perform malware detection with a very high detection rate and *no false positives*. In more details, the contribution of this dissertation is the description of a *new version* of AccessMiner, and how it can be implemented on top of our hypervisor-based framework.

The third, and last, contribution is the design and implementation of LynxFuzzer, a tool to automatically find vulnerabilities in Mac OS X kernel level components. LynxFuzzer, uses three different fuzzing engines to stress test kernel extensions (i.e., the dynamically loadable kernel components of OS X). LynxFuzzer leverages hardware-assisted virtualization to inspect tested kernel extensions, without inducing any change in them, a strong prerequisite of fuzzing: any inadvertent modification of the target, indeed, may easily lead to wrong results. LynxFuzzer has been implemented and tested on Mac OS X 10.8.2 (Mountain Lion), and provided unexpected results, individuating bugs in 6 of the 17 kexts we analyzed. Furthermore, both to support one of the fuzzing engine, and to give a partial evaluation of our fuzzer effectiveness, we implemented a code coverage analysis techniques that gave interesting results, in terms of obtained precision and overhead.

Recapping, we present solutions to three different systems security problems: transparent dynamic analysis, malware detection, and kernel fuzzing. Our dynamic analysis framework has been presented at the conference on *Automated Software Engineering* [39], and is complemented in this thesis with new parts that were not included in the original paper. The new version of AccessMiner has been presented during the *iCode Final Conference* and is now submitted to the *Journal of Computer Security* [38]. Finally, a paper on LynxFuzzer main contributions is still under submission, but the results of the code coverage technique we implemented have already been submitted as a short paper to the *Journal of Software Testing, Verification, and Reliability.*

1.2 Dissertation organization

The dissertation is organized as follows. Chapter 2 introduces some preliminary concepts on virtualization and program analysis. Chapter 3 presents the design and implementation of our generic, transparent, and completely dynamic, hypervisor analysis framework. Chapter 4 presents our new version of AccessMiner, and the results and performance overhead of using its *access activity models* to perform malware detection. Chapter 5 presents LynxFuzzer, a fuzzing tool that leverages hardware-assisted virtualization to automatically discover vulnerabilities in Mac OS X kernel modules. Chapter 6 compares our research work with related literature. Finally, chapter 7 concludes the dissertation and sketches some possible future research directions.

2 Preliminary concepts

This chapter contains some preliminary concepts that are at the very basis of this thesis. First, we give a glimpse at the many challenges of virtualization and, then, at how *hardware-assisted virtualization* can ease the cumbersome task of virtualizing an operating system. Then, we introduce concepts of *program analysis*, outlining both static and dynamic techniques, paying specific attention on their applications to systems security.

2.1 Virtualization

An *hypervisor* is a software entity that controls and regulates the execution of one or more *virtual machines*. For our purposes, we can classify hypervisors in two macro-categories: pure software and hardware-assisted.

The first category has been the *de facto* standard until the introduction of hardware virtualization technology and basically works by modifying the virtualized system to intercept and emulate a set of dangerous instructions. Such set includes every instruction that could allow the virtualized system to detect, or escape, its virtual container. Non dangerous instructions are executed natively, to boost up performances (as opposed to emulation [13], where every instruction is emulated).

Software virtualization is extremely complex to implement, and requires to face many challenging problems. For example, one is that the virtualized kernel cannot run at the privilege level it was originally designed for (typically, ring 0), because the hypervisor runs at the same level and this means that the virtualized system could escape its container. Thus, hypervisors use the *ring deprivileging* technique that runs the virtualized kernel in a less privileged level (ring 1, or ring 3, rather than the typical ring 0). Doing so, however, poses more problems, referred to as *ring aliasing* and *ring compression* [92]. The first includes all those instructions that behave differently, according to the privilege level in which they are executed, and that could let the virtualized kernel discover it is not being run at ring 0. The second derives from the fact that the memory management unit on an Intel architecture only distinguishes between two privilege levels (0 and 3), since paging structures have only 1 bit to control who can access a memory page [55]. This means that, when a software hypervisor uses ring deprivileging to reserve ring 0 for itself, and runs the virtualized kernel in ring 3, it deprives the latter of its main protection from user-space programs. Furthermore, ring deprivileging is not sufficient, as there are many other problems, related to both transparency and performances [92]. For example, some dangerous instructions are not privileged and cannot be intercepted with deprivileging (i.e., `sidt`), thus compelling the hypervisor to find other ways to intercept them.

Common pure-software virtualization techniques that deal with such problems are *paravirtualization* [11, 140] and *binary translation* [136, 137]. The first directly modifies the source code of the target kernel, for example to replace non-virtualizable instructions with others that explicitly transfer the execution to the hypervisor. The second requires to modify the target as well, but does so on kernel binaries, translating non-virtualizable instructions into sequences of virtual-machine-safe instructions with the same effect on the virtualized system [137]. For example, a `sidt` instruction could be translated into a sequence of instructions that fetches the base address of the interrupt descriptor table of the virtual machine, rather than the one currently stored in the `IDTR`, used by the hypervisor.

The second category includes hypervisors that leverage hardware support for virtualization, lately introduced in many commodity CPUs to greatly ease the task of virtualization, allowing hypervisors to spare themselves many of the intricacies of pure software approaches, as we will see in the following of section.

A further classification, orthogonal to the previous one, divides hypervisors between type 1, i.e., hypervisors that are directly installed on the bare metal, and type 2, i.e., hypervisors that are executed as part of an operating system already running on the hardware (e.g., a kernel module along with an user-space GUI).

2.2 Intel VT-x

All the approaches presented in this thesis strongly rely on hardware-assisted virtualization for their implementation, so in this section we give an overview of how such support helps creating transparent and extremely flexible hypervisors. Hardware virtualization technology available in Intel CPUs is called VT-x [92]. AMD technology, named SVM [4], is very similar and mostly differs in terminology. ARM recently introduced hardware support for virtualization as well [46]. In this thesis, however, we focus solely on Intel's technology.

Intel VT-x [56, 92] defines two new modes of operation for the CPU: VMX *root mode* and VMX *non-root mode*. The former is reserved to the hypervisor (also often referred to as Virtual Machine Monitor, or VMM[1]), while the guest (i.e., the system running *inside* a virtual machine) is confined in the latter. Software executing in both modes can operate in any of the four privilege levels that are supported by the CPU, eliminating the need for ring deprivileging techniques. Thus, the guest OS can execute at the highest CPU privilege level and the hypervisor can still supervise its execution without any modification.

In the typical deployment, the launch of a hypervisor consists of three steps. First, VMX root-mode is enabled. Second, the CPU is configured to execute the hypervisor in root-mode. Third, the guests are booted in non-root mode. However, Intel VT-x supports a particular feature, called *late launching of VMX modes*, that allows to launch a hypervisor at any time, thus giving the ability to turn a running OS into a guest (i.e., a system running inside a virtual machine). The procedure for such a delayed launch is the same as the one just described, with the exception of the third step. The state of the CPU for non-root mode is set to the exact same state of the CPU preceding the launch, such that, when the launch is completed, the execution of the OS and its applications resumes in non-root mode. The inverse procedure can be used to unload the hypervisor, disable VMX root-mode, and give back full control of the system to the OS.

Once the hypervisor is installed, the execution switches back and forth between non-root and root mode. In VT-x, such *transitions* are defined as *vm exit*, from the guest to the hypervisor, and *vm entry*, in the opposite direction. The hypervisor is executed only when particular events in the guest trigger an exit transition, and this is the only way a hypervisor can gain the control back, once it performs an entry. The set of *exit-triggering* events is extremely fine grained and can be configured by the hypervisor (e.g., it is possible to cause an exit whenever a particular *non-maskable interrupt* is triggered by the guest). Main exiting events include exceptions, interrupts, I/O operations, and the execution of privileged instructions (e.g., accesses to control registers). Every exit is handled by a dispatch function in the hypervisor that eventually performs an entry to give the control back to the guest.

Exits can also be *explicitly* requested by in-guest software, via `vmcall` instructions. Because of its similarity to system calls, this approach is commonly dubbed as *hypercall*. Indeed, an user-space program can fill appropriate CPU registers with the data it needs to send to the hypervisor and then execute a `vmcall`. Similarly to a `sysenter` instruction that causes a transition from user- to kernel-mode, this instruction causes a switch from non-root to root mode, transferring the execution control to the hypervisor. Once the request from non-root mode has been handled, the execution is returned to the user-space program, by the hypervisor, to the instruction after the `vmcall`.

The state of the CPU at the time of an exit and of an enter is stored in a data structure called Virtual Machine Control Structure, or *VMCS*. More precisely,

[1]In this dissertation we will use the terms *hypervisor* and *virtual machine monitor* (and its short form) to refer to software running in root mode, unless specified differently.

the VMCS stores the *host state*, *guest state*, *execution control fields*, *entry/exit control fields*, and *exit information fields*. Each of these fields has its own purpose:

- **Host state** stores the state of the processor that is loaded on exits to root mode, and consists of the state of all the registers of the CPU (except for general purpose registers).
- **Guest state** similarly stores the state of the processor that is loaded on entries to non-root mode. The guest state is updated automatically at every exit, such that the subsequent entry to non-root mode will resume the execution from the exact same status, unless the hypervisor alters it. Indeed, the hypervisor can change values of guest state fields to modify the state of a running guest at its will.
- **Execution control fields** define the behavior of the CPU in non-root mode and allow a fine-grained specification of which events should trigger an exit to root mode.
- **Entry and exit control fields** respectively govern particular configurations associated to entries and exits.
- **Exit information fields** are filled by the hardware whenever an exit occurs and contain a numeric code indicating the reason of the exit and many ancillary information. For example, in case of an exit triggered by an access to a control register (e.g., `mov eax, cr3`), such fields would include precious information, such as: the accessed register, the access type (read or write), the operand type (memory or register), the general purpose register (source or destination) and so on. It easy to see that such information are extremely valuable, because without the hardware support for virtualization, the hypervisor would have to revert to virtual machine introspection techniques [43], which are often onerous and error-prone.

As a last note on the VMCS, consider that its fields can be read and, in some cases, written by the hypervisor by means of `vmread` and `vmwrite` instructions, that can only be executed in root mode.

The last concept we need to introduce is that of extended page tables (EPT). Such technology has been introduced to support memory virtualization, the main source of overhead when virtualizing a system. It basically consists in a paging structure for a full 64 bit address space (i.e., a 4 level paging structure) [56] that keeps a mapping between guest and host physical addresses. When such mechanism is enabled, the standard *virtual-to-physical* address translation of guests is modified as in Figure 2.1.

When software in the guest references a virtual address, this is translated into a physical address by the memory management unit (MMU), as normally happens on non-virtualized systems. Such an address however, is not a real

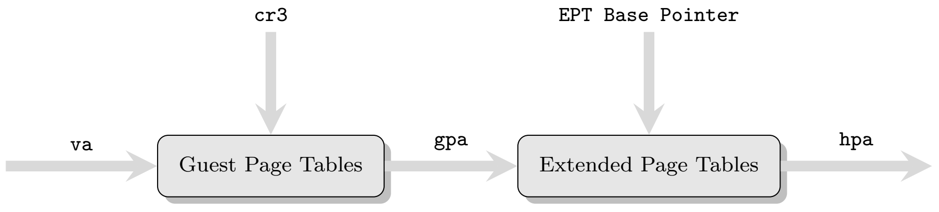


Figure 2.1: Address translation with EPT

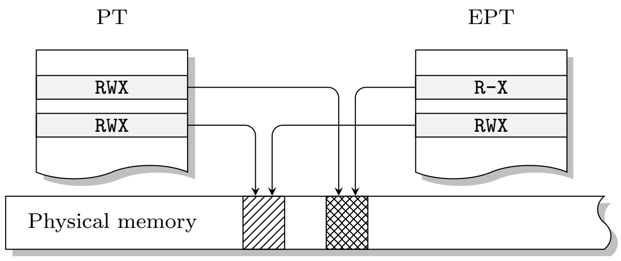


Figure 2.2: Physical page protection through EPT structures

physical address, but rather a *guest physical address* (*gpa*). The hardware then, similarly to the MMU, walks EPT paging structures to translate a *gpa* into a *host physical address* (hpa), that corresponds to the actual physical address in the RAM. EPT technology also defines two new exits: *EPT Misconfiguration* and *EPT Violation*, respectively caused by wrong settings in EPT paging entries and by the guest attempting to access memory areas it is not allowed to. By altering EPT entries, indeed, the hypervisor has full control on how the guest accesses physical memory. An example of this can be observed in Figure 2.2, where ▩ denotes pages protected through EPT. In this case, any write-access by the guest to that page triggers a violation. Exits information fields of the violation are filled with extremely useful information, such as the accessed *gpa*, its corresponding linear address, and the type of attempted access (R, W or X).

The same protection approach can be implemented purely via software, with *shadow page tables*. The idea is the same, with the difference that the hypervisor must directly modify entries of the page table. This, however, entails a lot of problems, mainly due to the fact that the guest can access the page table as well, thus the hypervisor must protect both the page *and* the page table, to prevent the guest from detecting or disabling the protection. Another problem is that, while EPT allows to work directly on physical memory pages, altering a page table entry corresponds to altering the physical-to-virtual mapping of a single process. Altering an entry, thus, protects the virtual mapping of the page, rather than the page itself and nothing prevents the guest from creating a new mapping to the protected page, bypassing the protection. Thus, more complications arise, because the solution to this problem requires that every write access to any PT of the guest system must be mediated to prevent re-mapping of protected pages. Shadow page tables were the standard protection scheme of both software and hardware hypervisors, before the introduction of EPT technonlogy. The use of EPT allows a hypervisor to drastically reduce its overhead, as memory virtualization is one of the main source of overhead in software hypervisors [30].

2.3 Program analysis

Program analysis usually refers to techniques that automatically extract properties of a computer program. Such techniques are mainly used in the field of compilers, to produce extremely optimized and efficient code [3]. However, they are important in other research fields as well, among which we find systems security. Consider the following example: it is well-known that it is impossible to formally proof that a program is free of bugs, as this problem is equivalent to the halting problem [134]. Program analysis, however, can greatly help developers to find bugs in their applications and make them better and safer.

Program analysis can be conducted either statically or dynamically and both solutions have their perks and disadvantages. During years, however, this division began to fade, as researchers started to leverage strong points of both, giving birth to many hybrid techniques.

2.3.1 Static analysis

Static analysis examines the code of a program (either source or binary) to extrapolate information on its properties and on every possible behavior that can be exhibited. Static analysis is *complete*, as the properties it observes do not anyhow depend on the input that the program receives, or on the configuration of the analysis environment. Techniques following this approach are commonly adopted by compilers that analyze the source code of the software to apply many compile-time optimizations [31].

Static analysis techniques are also employed in security, especially in malware analysis and detection. When dealing with malware, indeed, being able to extract information without actually running a potentially dangerous program is extremely useful. Particular instruction or byte sequences, characteristic patterns in a program Control Flow Graph [3, 16], and opcode n-grams [50] are just examples of how information extracted through static analysis have been used for malware detection and analysis.

Such techniques, however, suffer some limitations due to their static nature. Indirect control transfer instructions (e.g., `jmp *eax`) cannot be followed statically, unless the value of `eax` is easily deducible. This may result in missing the analysis of some portions of code (for example, code that can be reached only through an indirect CTI may be ignored by a disassembler). Furthermore, malware authors introduced a plethora of techniques to thwart static analyses, such as obfuscation, packing, polymorphism, and metamorphism [72, 115, 77, 91, 129, 66].

2.3.2 Dynamic analysis

Dynamic analysis is completely different from its static counterpart: indeed, it executes the target and observes its behavior and properties at runtime. In such scenario, some observations become extremely simpler. For example, getting the value of a particular variable at a given point during execution just requires to execute the program up to that point and observe its value. Unfortunately, this automatically implies that each observation conducted through dynamic analysis is strictly related to the execution that led to it. This means that each observed value and behavior depends on the program's input and on the analysis environment. Practically speaking, this may results in missing some properties that would have been observed through static analysis. The classical, security-related, example in which this limitation could badly affect the results of dynamic analysis are logic bombs (i.e., malware that executes its payload only if certain conditions are met, typically a date and time). If a malware detector runs a logic bomb, and its triggering conditions are not met, payload is not executed and the detector may incorrectly classify the malware as benign.

Independently from their implementation and granularity level (e.g., instructions, system calls, library functions), dynamic analysis has long been used for malware analysis [35], since techniques to hinder static analysis commonly employed by malware do not affect it. Such approaches, however, are not bulletproof either. The main countermeasure adopted by malware is to try to detect the presence of an analyzer and, if needed, hide any malicious behavior [102, 110, 22]. This led to many efforts to create new environments that could grant more transparency, and soon resulted in a "race to the lower level" between malware and analysis tools [109, 33].

Despite these problems, the possibility to observe how a program behaves at run time has been used to tackle many security problems, not limited to malware analysis [35]. Examples of problems that have been tackled through dynamic analysis are: intrusion detection [43], exploit detection and prevention [15, 94], automatic exploit generation [51, 8], automatic identification of vulnerabilities [19, 68].

2.3.3 Symbolic and concolic execution

Symbolic execution is a testing approach that, rather than feeding concrete inputs to a program, uses symbolic inputs [60, 18]. Each time the target program uses an input, this gets replaced with a symbolic value, and every operation that works on input is modified to work on symbolic inputs. When a branch whose condition depends on the input is met, the condition is stored and the symbolic execution engine follows both true and false paths. The execution terminates when the a particular point (or the end) of the target is reached. Using a constraint solver [89] on the set of conditions encountered during execution, the analyzer can generate a set of concrete input values that lead the execution of the program to the point where the analysis stopped. It is clear that this approach is extremely helpful for many kind of analyses, since it basically solve the main problem of dynamic analysis (i.e., observations are strongly bounded to what is executed). Consider once again the logic bomb example: symbolic execution would allow to automatically extract the triggering conditions of the malware. Unfortunately, symbolic execution analysis techniques suffer from the *path explosion* problem. In complex programs, where the number of possible execution paths is enormous, a symbolic analysis engine would require too much time to follow all of them and generate corresponding concrete input values.

An evolution of symbolic execution that aims at maximizing the code coverage of an analyzed program is *concolic execution* [114, 76]. With this approach, the target is, at first, executed on concrete inputs. Concurrently, the symbolic engine follows the same execution path but threats variables as symbolic and keeps track of branch conditions. When the execution is over, the constraint solver generates a new set of concrete inputs, starting from the previous one, that will lead the execution down a different path. Concolic execution does not solve the path explosion problem on its now, but rather helps getting the execution down long and intricated paths faster.

Many efforts have been put into trying to solve the path explosion problem, e.g., symbolically executing only portions of the target [24], or reducing the amount of variables that must be treated as symbolic [49].

3 Dynamic analysis through hardware-assisted virtualization

Dynamic analysis techniques can be roughly classified in two main categories: those that leverage user- or kernel-level components and those that leverage emulation and virtualization to work "out-of-the-box".

Techniques belonging to the first category are usually extremely convenient when dealing with user-level, non privileged targets. There are many successful tools and frameworks that are well-suited to dynamically analyze user-space targets, by either leveraging system libraries (e.g., gdb [121]), or binary instrumentation (e.g., PIN [73], Valgrind [93]). When dealing with the analysis of kernel-level targets, on the other hand, such "in-the-box" approaches are much more complicated and often require *a priori* modifications of the target kernel (e.g., Dtrace [21]), or to load new components that dynamically modify the target itself (e.g., KernInst [131]), to install hooks that will be later used by the tools to intercept events of interest. These approaches work well to trace the execution of targets (both at user- and kernel-level) that do not try to detect or escape the analysis environment and whose behavior is not influenced by the environment itself (think, for example, of a buggy program that only exhibits a faulty behavior when *not* being debugged). Unfortunately, these conditions do not always hold, especially when dealing with malware [35], or benign software that do not want to be analyzed [32].

Out-of-the-box approaches strive to solve limitations of their in-the-box counterpart but, unfortunately, suffer from some issues as well. They are based on running the target as a guest of a virtual machine, and then performing the analyses from *outside*, i.e., from the hypervisor [43, 25, 62, 34, 12, 106, 144, 78]. Unfortunately, problems of such out-of-the-box approaches are manifold. First, they require the target to be *natively* bootstrapped as a guest of a virtual machine, and this is not always possible, especially when dealing with production systems. Second, the target may require to interact with particular hardware devices to work properly, and emulation of uncommon hardware is often unavailable in commodity hypervisors and must be implemented manually. Last, but of no lesser importance, the transparency of software virtualizers and emulators has been brought into question by the discovery of many so-called "red pills" [110, 80, 81, 102]. A red pill is a piece of software that behaves differently when executed on a real or emulated CPU. Malware authors embed such code snippets into their malware to detect the presence of analysis environments and consequently hide any suspicious behavior.

In this chapter we present some improvements over a framework we presented earlier [39]. In particular, we proposed a *framework that brings together the advantages of both approaches: it can be used on commodity production systems (i.e., off-the-shelf products, whose source code or debugging symbols are not necessarily available), since it does not require to instrument the system under test, and it is able to inspect systems running on real hardware, since it does not require an emulation container* [39]. Thanks to its features (*fully dynamic*, *transparent*, *loosely dependent on the target operating system*), our framework can be used to analyze any *running* production system, without any need to modify it. Moreover, since the framework itself is not accessible from the target, its presence is hard to be detected by malicious code. To obtain such desiderable features, the framework leverages hardware support for virtualization available on commodity x86 CPUs [92, 4], that we illustrated in section 2.2. Furthermore, to enable the analysis of *running* operating systems (a strong limitation of software virtualization-based tools) we exploit a feature of the hardware that allows to install an hypervisor and to *migrate a running system into a virtual machine* at runtime. This feature is part of hardware support for virtualization and is known as late launching.

In this chapter we illustrate the internals of our framework, that constitutes the basis of all the work presented in this dissertation. Furthermore, we discuss the improvements that we made to our framework. These improvements include the support for extended page tables (EPT) and EPT based watchpoints and breakpoints, along with the implementation of some virtual machine introspection techniques for Windows 7 and Linux guests.

3.1 Overview of the framework

One of the most peculiar features of our framework is its ability to be installed on live system that do not natively provide any pre-existing software support (e.g., hooks in the kernel). Indeed, it operates as a minimal hypervisor and only requires a CPU with hardware support for virtualization (that is nowadays present on most Intel [92] and AMD [4] CPUs, as well as on some ARM architectures [46]), which we presented in section 2.2. Differently from other hardware-assisted hypervisors [11] that require to be the first piece of software launched during a machine boot, we leverage *late launching* to temporarily *migrate a running op-*

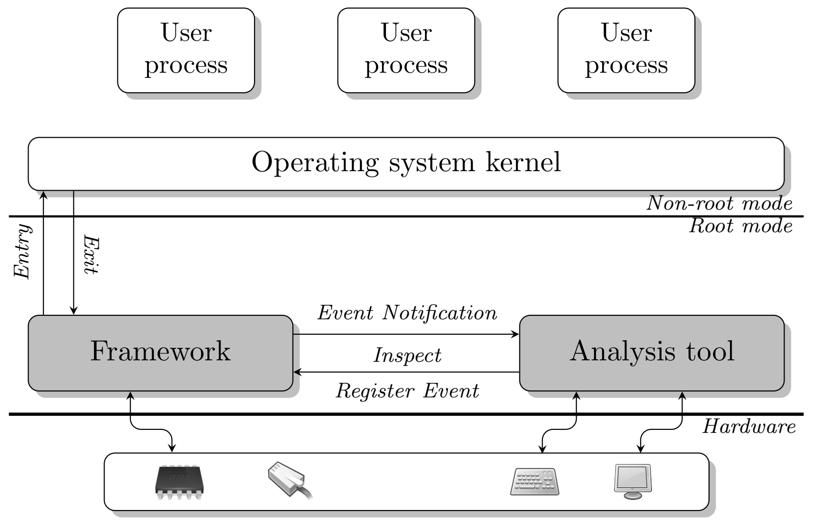


Figure 3.1: Overview of the framework

erating system in a virtual container.

Practically speaking, we do not move, or modify, the running operating system at all. On the other hand, by enabling hardware support for virtualization, the execution environment is extended with the two operating modes that we illustrated in section 2.2: *root mode* and *non-root mode*. The running operating system is associated to non-root mode, while the hypervisor is associated to root mode. By doing so, the operating system and its applications actually become a guest of a virtual machine, under the control of our hypervisor framework. Similarly, the hypervisor can be unloaded, and the original mode of execution of the operating system restored, by simply disabling hardware support for virtualization.

A similar approach has been used in Eudaemon, by Portokalidis and Bos [105], a system to detect and block exploitation attempts. Eudaemon can attach to a running process and literally migrate it into an user space emulator, where it will be dynamically analyzed. Once the analysis is over, possibly detecting and blocking an exploit attempt, the eexecution can be seamlessly return to the native binary. Our late launching based approach is conceptually equivalent, but while user space emulation is sufficient for Eudaemon, we leverage hardware assisted virtualization to transparently migrate a whole operating system under the control of our framework.

As soon as our hypervisor finishes loading, the guest continues to run unaffected, except for the fact that its execution might be interrupted by an exit to root mode. Being executed at the highest privilege level, the routine that handles the exit has complete read/write control of the state of the guest system (of both memory and CPU registers).

Figure 3.1 depicts an overview of our framework. As we can see, the framework operates in root mode and exports an API that allows to build custom tools that

Event	**Exit cause**	**Native exit**
`ProcessSwitch`	Change of page table address	√
`Exception`	Exception	√
`BreakpointHit`	Debug except. / Page fault except.	
`WatchpointHit`	Page fault except.	
`SyscallEntry`	Breakpoint on syscall entry point	
`SyscallExit`	Breakpoint on syscall return address	
`IOOperationPort`	Port read/write	√
`IOOperationMmap`	Watchpoint on device memory	

Table 3.1: Main traceable events

interact with it to dynamically analyze the target system. The framework itself does not perform any analysis; rather, it acts as a bridge between the target system and the analysis tool. This greatly eases the task of creating tools that operate in root mode, hiding all (or most of) the intricacies of working in this execution mode.

As we specified in section 2.2, a hardware assisted hypervisor gets executed only when the guest OS triggers an *exit* to root mode. Our framework takes care of intercepting and handling every exit triggered by the guest. Furthermore, it also creates a high level representation of the low level event that triggered the exit, possibly sending a notification of this event to an analysis tool. Analysis tools built on top of our framework may indeed require to be notified when particular events happens. A subset of the most important events that can be intercepted is reported in Table 3.1. Each entry of the table distinguish between native events, i.e., those that have a corresponding exit, and others that are usually created on top of one or more low level ones. An example of native event is `ProcessSwitch`, that is directly associated to exits triggered by a write operation on the `cr3` control register (that contains the base address of the paging structures of a process). More complex events, such as `BreakpointHit`, are derivation of other events. For example, when using software breakpoints, the `BreakpointHit` event is a particular case of `Exception`, while the `SyscallEntry` event is a particular case of `BreakpointHit`.

As a final remark on events, please note that only some exits are conditional while others are not. This means that the framework is always dealing with a small set of exits that are unconditionally triggered, as those triggered by an execution of the `cpuid` instruction, while others can be fine-granedly tuned according to the need of the analysis tools. For example, exits can be enabled for single exceptions types, and the framework can also control which exits to enable dynamically, turning them on and off when needed. This allows our framework to be extremely flexible and to avoid useless overhead.

Just like the framework is executed only when the guest triggers an exit, an

Function	**Parameters**
Tracing	
`SetBreakpoint`	`cr3`, Address
`SetWatchpoint`	`cr3`, Address, Write/Read
`RegisterEvent`	Event type, Event condition, Handler
Inspection	
`MapPhysicalPage`	Address
`UnmapPhysicalPage`	Address
`ReadPhysicalRegion`	Address, Size
`WritePhysicalRegion`	Address, Size
`GetPhysicalAddress`	`cr3`, Address
`IsAddressValid`	`cr3`, Address
`ReadVirtualRegion`	`cr3`, Address, Size
`WriteVirtualRegion`	`cr3`, Address, Size
`GetRegisters`	-

Table 3.2: Main API Functions

analysis tools is invoked only by the framework. To make it so, a tool must register for at least one event. Once it gets executed, it can leverage the API of the framework to enable/disable the tracing of events, or to inspect the state of the guest. A brief summary of the main API functions and their parameters is reported in Table 3.2.

API functions can be divided into two main branches: those for the tracing of particular events and those to inspect the state of the guest once an exit is triggered. So, for example, to use breakpoints, an analysis tools would at first call `RegisterEvent` to register a handler for the `BreakpointHit` event, and then invoke the `SetBreakpoint` to place a breakpoint on the desider address. Note that this function, along with many others that deal with virtual memory, requires to specify a `cr3` value as a parameter. This is because the framework needs to know in which address space of the guest to operate. Indeed, the `cr3` register always points to the base of the paging structures of the currently executing process and, as we will see in section 3.2.3, this information is essential for the inspection of the guest. If the analyzed guest is supported, however, it is possible to use the OS-dependend API offered by our framework, and explained in section 3.2.4, to retrieve the value of the `cr3` for a particular process directly using its name.

3.2 Implementation details

In this section, we introduce some implementation details of our framework, and explain if and how they were improved.

3.2.1 Framework loading

The framework, along with any eventual analysis tool, is loaded by a minimal kernel driver. Indeed, the initial operations we need to perform require to operate in kernel mode.

We took extreme care in creating a driver as small and simple as possible, to reduce to a minimum our influence on the running kernel. Once the loading is over, furthermore, the whole driver can be removed. When VMX modes are enabled, the VMCS is made accessible initially to the loader, and subsequently, when the loading is completed, only to the framework. The main tasks of the loader are to enable VMX modes and to configure the VMX data structure such that the operating system and user-space applications continue to run in non-root mode, while the framework and the analysis tool are executed in root mode. This is obtained by configuring the *guest state* fields of the VMCS to perfectly reflect the state of the guest OS just before switching to root mode for the first time.

Then, the driver prepares the *host state* fields for the launch, creating a private address space for the hypervisor, and setting the address of the exit handler function as its entry point. This function will work as a gate, receiving all exits and dispatching them to the corresponding handlers.

Finally, the loader has to configure the CPU such that all the events necessary for the tool to trace the execution of the system trigger exits to root mode. This set of enabled events can be later modified according to the need of the analysis. When the initialization is completed, the driver unloads itself and resumes the execution of the system.

To make our framework robust, we must be sure that none of its memory regions can be accessed by any component of the guest operating system. As we said, during the initial loading phase, the VMCS is configured to grant the hypervisor a private address space. However, this is not enough to protect the framework from inferences of the guest operating system. Indeed, a kernel module running in the guest could map any physical address by directly modifying the paging structures of the guest and, thus, read any part of the hypervisor (assuming it could get a leak to where the hypervisor is). To prevent this, our framework includes a further protection layer, based on *shadow page tables* [117]. A shadow page table is a copy of a real page table that is used to create a virtual-to-physical translation different than the mapping contained in the original page table. In particular, the hypervisor contains a shadow page table for each page table on the guest system, and keeps a 1:1 mapping between these two tables, except for entries mapping pages that the guest cannot access. This protection mechanism has long been used by hypervisors, and it is the only solution if extended page tables technology is not available [56]. However, as we discussed in section 2.2, this introduces a great overhead in our framework. Indeed, the protection mechanism is based on intercepting two events: context switches (`ProcessSwitch`), and page faults (`Exception`). The first is required to "replace" the page table of each process with the patched one. The second is required because we need to intercept attempts to map physical memory areas of the hypervisor, so we must trap whenever software running in the guest accesses a page table. Thus, we modify each page table entry that maps a page table itself, so that page tables are nor readable nor writable. By doing so, each time the guest accesses page tables, a page fault exception is thrown and intercepted by the hypervisor. For each fault, the framework checks if the guest is trying to map a restricted memory area, or if it is reading one of the modified page table entries. In both cases, the hypervisor alters the operation to prevent software running in the guest from accessing its memory areas, or to detect the presence of altered page table entries.

To prevent the overhead due to the use of shadow page tables, we improved our framework to use extended page tables (EPT), on systems with such support. As we explained in section 2.2, EPT allow a hypervisor to directly control how the guest accesses physical memory. As soon as the hypervisor is loaded, indeed, it takes care of identifying where it has been placed in physical memory by the loader module. Then, to prevent the guest from accessing this areas, it alters the corresponding EPT entries so that every access to these area is redirected on a different page. It is clear to see how this mechanism is extremely more efficient: there is no more need for the hypervisor to intercept either `cr3` writes or page faults. Furthermore, we will see in section 3.2.2, EPT plays also a key role in the implementation of efficient, and completely transparent, watchpoints and breakpoints. At load time, the framework sets EPT paging structures to create a 1:1 mapping between every guest-physical and host-physical address, except for entries corresponding to the hypervisor, that are made inaccessible to the guest.

3.2.2 Guest tracing

Despite being commonly used, and a well established concept in debugging, breakpoints and watchpoints are two of the most complicated events to implement in root mode. Most CPUs provide hardware breakpoints and watchpoints, that are easy to use and transparent. By using a set of debug registers, it is indeed possible to cause a debug trap whenever a virtual address is accessed, either in read, write, or execution, according to the registers configuration. Unfortunately, such registers are limited in number: only 4 are available (`DR0,..., DR3`) to specify on which address to place a breakpoint/watchpoint. Furthermore, these registers are shared between the hypervisor and the guests. Therefore, they cannot be used simultaneously by the analyzed system and by the framework.

To bypass this problem and allow an arbitrary number of breakpoints, our framework originally used *software breakpoints*. Software breakpoints are usually implemented by replacing the first byte of the target instruction with a one-byte instruction that triggers an exception when executed. In particular, we implemented software breakpoints using a standard technique, i.e., by replacing the byte at the address on which a breakpoint must be set with an `int 3` (`0xcc`).

When the CPU executes such instruction, it triggers a `#BP` exception that can be easily intercepted by the hypervisor. The corresponding exit is handled by the hypervisor restoring the byte of the original instruction and possibly notifying the event to an analysis tool. Then, if the breakpoint is not persistent the execution of the system is resumed. Otherwise, we configure the guest to perform a *single-step*, by raising the Trap and Resume flags in the `EFLAGS` register of the guest (altering the value stored in the VMCS). By doing so, the guest will execute the faulty instruction, that no more triggers a `#BP` because it has been restored but, after the CPU executes it, a debug exception (`#DB`) is triggered, causing an exit. Indeed, the hypervisor is configured to intercept such exceptions, and handles them by replacing once again the instruction with a software breakpoint then continuing the execution of the guest.

We use this technique to avoid the need for emulation: without it, all faulty instructions would need to be emulated. Using single stepping, instead, we can avoid emulation, but this is not always possible as some instructions still need to be manually handled. For example, consider a `pushfd` instruction: single stepping over it causes the pushed `EFLAGS` to include the single step flag, meaning that the first following `popfd` restores this wrong value, possibly modifying the behavior of the target. During the implementation of our framework, we found a set of instructions that cannot be easily single-stepped on and included emulation procedures for each of them in our hypervisor. When stepping, we also lower the interrupt flag, so that interrupts cannot transfer the execution flow out of the target before we can re-enable the protection.

Clearly, the approach to breakpoints we just described is not transparent for the analyzed system (i.e., a malicious program could spot a software breakpoint and alter its behavior). An alternative and transparent approach is to use the same technique we use for watchpoints, as described in the next paragraph. Our framework supports both approaches.

In our framework, watchpoints are based on shadow page tables. As we said in section 3.2.1, such mechanism allows to define custom protection for any memory location of the guest, thus, we leverage it to implement transparent watchpoints [135]. More precisely, when we need to place a watchpoint on a given memory address, we alter its entry in the paging structures of the guest to remove access permissions (either read or write) according to the kind of required watchpoint. By doing so, as soon as the guest tries to access that memory area, a page fault (`#PF`) is triggered. As for `#BP`, `#PF` exceptions can be easily intercepted by the hypervisor, but they require a more careful handling. More precisely, to set a watchpoint we need to remove access permissions from a whole page, thus not every intercepted `#PF` automatically implies that a watchpoint has been hit. Each `#PF`-triggered exit is inspected by the hypervisor, to check whether the faulty address corresponds to a watchpoint. If this is the case, the framework delivers the event to the analysis tool, otherwise (or if the watchpoint is permanent), we use the same single-stepping approach that we illustrated for software breakpoints.

Obviously this approach increases the run-time overhead, due both to the huge number of synthetic page fault exceptions it generates and to the interception of every page fault that is triggered by the guest; however, it also guarantees a higher level of transparency to both the guest operating system and user-space applications.

If available, extended page tables can be used to implement watchpoints. In particular, the same approach based on shadow page tables can be implemented leveraging EPT, with the only difference that we need to work on physical rather than virtual addresses (recall that the mapping controlled by EPT paging structures is between guest and host physical addresses). Thus, at first the framework translates the virtual address on which the breakpoint must be set, by using guest memory handling primitives (explained in section 3.2.3). Then, it alters the appropriate EPT entry to remove permissions as required by the operation to be performed (i.e., execution for breakpoints, read and/or write for watchpoints). This causes the triggering of an EPT violation as soon as the guest attempts to access a protected physical page, according to the removed permission. The exit handling routine of the framework checks if the faulty address matches one on which a watchpoint was set—this is not always the case, since we work at a page-level granularity—and possibly delivers the event. This alternative approach is much more efficient than its software counterpart: there are no artificial events (`#BP` or `#PF`) to be masked and paging structures of the guest must not be modified (consequently removing the need to mask this changes for transparency). However, working at a lower level (i.e., physical memory) rather than directly modifying page tables requires special care. For example, if a breakpoint is placed on a physical page that is shared between two or more processes (e.g., a shared library), each one of them will trigger a violation when accessing such page. Thus, the framework must identify the process that is triggering the violation, to allow tools built on top of our framework to filter watchpoint and breakpoint events according to the triggering process.

As we briefly mentioned earlier, many other higher-level events, such as system calls, are traced through breakpoints. For example, to trace system calls we put a breakpoint on the entry point of the system call gate [33].

3.2.3 Guest inspection and manipulation

The framework provides API functions to inspect and manipulate the state of the guest operating system. In particular, we allow to inspect the registers and the memory of the guest. Inspecting registers is trivial: some of them are automatically stored into the VMCS whenever an exit occurs, while others are saved by the exit handling procedure of our framework. Functions are provided to analysis tools to either read or write registers; writes will be reflected in the guest when performing an entry.

Inspection and manipulation of the memory of the guest are, unfortunately, more complex. With paging, virtual addresses are translated by the MMU into physical addresses according to the content of the page table and direct physical addressing is not possible. The page tables of a process can always be accessed through the `cr3` register, that points to the base of the first level paging structure (the Page Directory, on a 32 bit system). During a context switch, the `cr3` register is swapped, so that every process has its own private address space (although, some parts may be shared with other processes). This mechanism is essential in every modern operating system, but it poses a problem for our framework. Even if we operate in root mode, indeed, we are constrained to operate with virtual memory addresses and can only access those memory areas that are mapped in our own address space. Nonetheless, we want to directly access physical memory as well as the virtual address space of any process running on the guest system. That is to say, we want to "navigate" into the virtual memory of any process, regardless of the actual value of the `cr3` register. The most intuitive solution, that we use in our framework, is to implement via software the same address translation mechanism that is used by the MMU, so to use it with arbitrary `cr3` values. Unfortunately, such register and paging structures always refer to physical addresses and, thus, we still must rely on a procedure to read physical memory. To overcome this problem, during the loading phase, the module sets up the private address space of the hypervisor so that it can access paging structures at a fixed virtual address. This is a common setup for many operating systems; for example, windows sets a virtual-to-physical mapping of paging structures in the virtual address range `0xc0000000:0xc0400000`. Once this mapping is created, the loader module sets the `cr3` that will be used by the hypervisor in the `host_cr3` field of the VMCS. As a consequence of this setup, when in root mode, it will be possible for our framework to alter its own paging structures by only referring to virtual addresses.

Each time the framework needs to read (or write) a physical page, it modifies its paging structures to map it into the first unused virtual address. With a primitive to read from physical addresses, then, it is straightforward to implement the aforementioned software MMU and, consequently, read from any virtual address space of the guest. On top of this mechanism, we implemented all memory-related inspection functions listed in Table 3.2.

3.2.4 OS-dependent functions

Every function in the API provided by our framework is generic, and it works regardless of the guest operating systems. In some situations, however, it may be useful to have some high level information at avail. For example, every inspection function of our framework that deals with virtual memory addresses requires a `cr3` value as parameter, to indicate which virtual address space to inspect. This piece of information, by itself, is enough to univocally refer to a process in the guest system. For a user of our framework, however, using this 32-bit identifier

Name	**Description**
`GetFuncAddr(`*n*`)`	Return the address of the function n
`GetFuncName(`*a*`)`	Return the name of the function at address a
`GetProcName(`*cr3*`)`	Get the name of process given its *cr3*
`GetProcPID(`*cr3*`)`	Get the PID of process given its `cr3`
`GetProcByName(`*n*`)`	Get info (PID, `cr3`, ...) of process with name n
`GetProcList()`	Enumerate processes
`GetDriverList()`	Enumerate device drivers
`GetConnectionList()`	Enumerate network connections

Table 3.3: Main functions of the OS-dependent API

to refer to a process may be cumbersome, as it is much more natural to use its name, or its PID.

For this reason, we decided to include a set of OS-dependent functions in our framework, some of which are summarized in Table 3.3, to make the analysis of certain guests more comfortable. All of these functions rely on virtual machine introspection techniques (VMI) [43] to analyze the internal structures of the guest operating system to translate OS-independent information into something more user-friendly. In the following of this section, we give some details of the OS-dependent API of our framework.

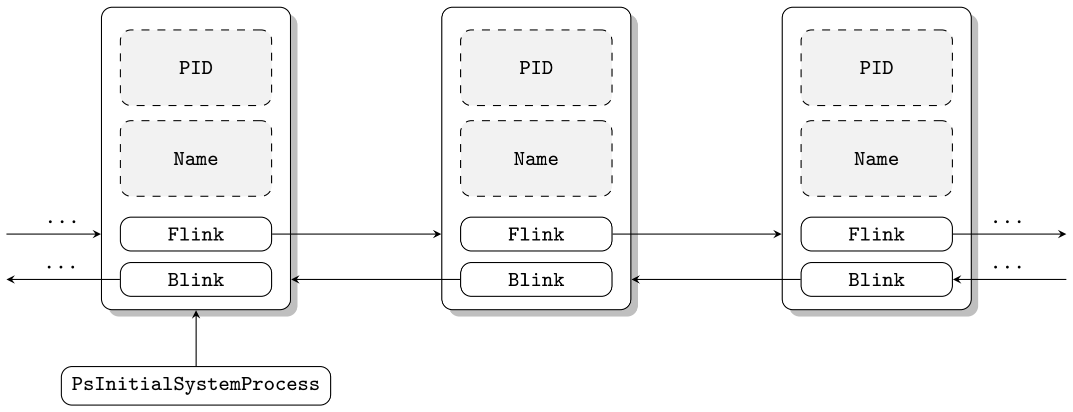


Figure 3.2: Process list in the guest memory

Figure 3.2 reports the typical layout of the kernel structures that contain information on processes. In the Figure we use names from the Windows kernel, but the structure is almost equivalent on Linux. All the functions in the `GetProc` family inspect this structures to extract information on processes. As it is possible to see, parsing these structures is quite straightforward: being a circular double-linked list, the framework starts from the first process and just keeps following the forward link until it gets back to the beginning. Each structure contains a plethora of information about each process (such as its name, its PID, and so on).

To find the first entry of the linked list, we leverage either symbols (`PsInitialSystemProcess` on Windows) or exported variables (`init_task` on Linux). As an alternative, when such starting points are not available, we revert to other techniques. On Windows 7, `%fs:0x20` always points to the base of the kernel processor control region (`KPRCB`). The `KPRCB` contains a pointer to the currently executing thread that in turn points to the `EPROCESS` structure of its owner process. This structure is an entry in the linked list of processes, so we can reach every other process starting from this entry. A similar technique can be used for Linux guests, with the difference that we retrieve the pointer to the `thread_info` structure of the currently executing thread by masking out the lowest 13 bits of the kernel stack pointer. From there, just like on Windows, it is possible to retrieve a pointer to the `task_struct` of the current process and navigate the linked list from there.

Network connections are not as friendly to inspect as processes and drivers, since their implementation strongly differs in Linux and Windows. Introspecting network connections of a Linux guest is quite straightforward, since Linux follows the well established UNIX concept that "everything is a file". Indeed, each process that makes use of a network connection has a file descriptor associated to each open socket. The hypervisor can inspect open file descriptors of a process by looking at the same structure used by the `GetProc` family functions. In particular, the process-related structure contains pointers to a list of other structures, containing information on open file descriptors. The `GetConnectionList()` function inspects the open file descriptor list for each process. Each entry in the list is then inspected to determine which one corresponds to a network connection. Entries identified as such contains a wealth of information, such as the network protocol (e.g., `TCP`, `UDP`), the state of the connection (e.g., `ESTABLISHED`, `LISTENING`), local and remote ports, and so on.

Extracting network connections from a Windows guest is more complex, and the required introspection techniques vary greatly according to the Windows version. Contrarily to Linux, on Windows network connections are stored centrally, inside the `tcpip.sys` driver. Under Windows XP, this driver contains tables with an entry for each connection that specifies local and remote ports and addresses, plus the PID of the process that opened the connection. Under Windows 7, `tcpip.sys` internal structures are not so straightforward, as they include multiple layers of tables and hashtables that must be carefully parsed to extract network connection information. Given the great complexity of the structures involved in this introspection technique, we do not linger on details of their parsing and refer to [104] for further details. As was the case for Linux guests, where we relied on the `GetProcList` function, even in Windows guests we need to rely on other OS-dependent features. In particular, we need to find the address where `tcpip.sys` is loaded in memory, since this is the base to find its internal structures, and this can be done using functions to retrieve information on loaded modules.

Finally, note that some OS-dependent functions, such as `GetFuncName` rely on the availability of debugging symbols for the analyzed guest. If symbols are not available, such functions are automatically disabled, while others, that purely rely on VMI, are unaffected. If the guest operating system is not supported, however, OS-dependent functions are disabled, and only OS-independent functionalities are available. Our current implementation offers an OS-dependent interface for: Windows XP, Windows 7, and for some Linux kernel versions.

3.3 HyperDbg: a Kernel Debugger

The first analysis tool that was built on top of our framework is HyperDbg, an interactive kernel debugger. The current implementation of HyperDbg supports Microsoft Windows XP, Windows 7, and Linux. We released the code of HyperDbg, along with the code of the framework, under the GPL (v3.0) license. The code is available at:

`http://code.google.com/p/hyperdbg/`

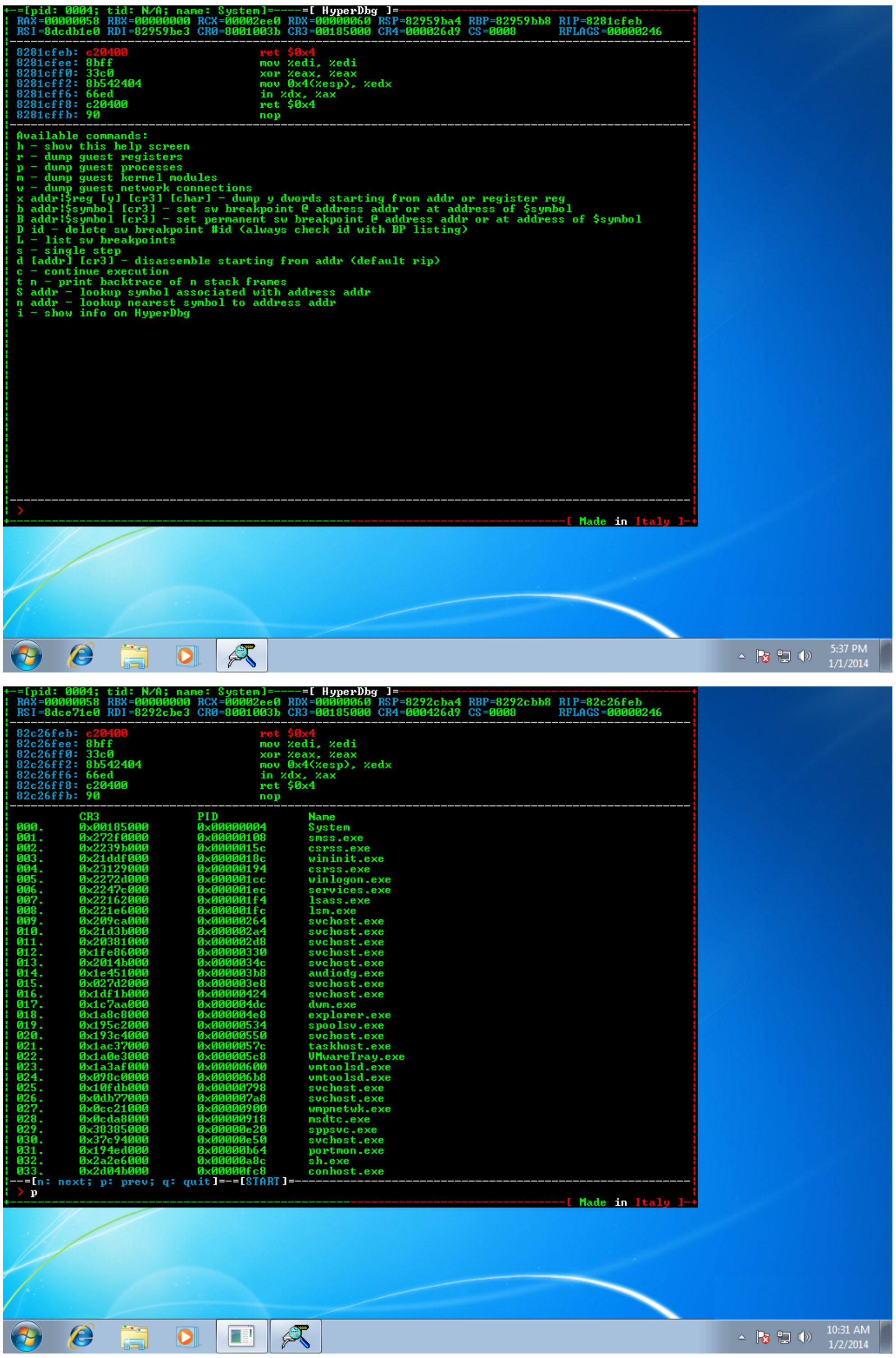
We recently ported the core of our framework to 64 bit Intel architectures and implemented an additional loading driver for Mac OS X, but these parts have not been released yet.

Thanks to the fact that it is built on top of our framework, HyperDbg is run in root mode, it is OS-independent and grants complete transparency to the guest operating system and its applications. The debugger has a very rough graphical user interface that is activated either when the user invokes the debugger by pressing a special hot-key, or when an event occurs that requires the attention of the user (e.g., a breakpoint is hit). Once in root mode, the user interacts with the debugger to perform several different operations, such as setting breakpoints, single stepping, inspecting the state of the guest, and so on.

Figure 3.3 shows HyperDbg in action under a Windows 7 guest. In particular, the first screenshot shows the list of commands available to HyperDbg users. Every one of them is built on top of the API that we presented in Table 3.2, or leveraging OS-dependent functions (section 3.2.4). Such functions are also used to facilitate the debugging. For example, the second screenshot in Figure 3.3 shows the list of the processes in the guest.

Since HyperDbg is run on top of our hypervisor framework, it can be used to debug *any* part of the guest, including critical components such as the process scheduler, or interrupt handlers.

The remaining of this section describes how we used the facilities of our framework to implement the user interface and the component to receive commands from the user.



```
+--=[pid: 0004; tid: N/A; name: System]=----=[ HyperDbg ]=------------------------------------------+
! RAX=00000058 RBX=00000000 RCX=00002ee0 RDX=00000060 RSP=82959ba4 RBP=82959bb8 RIP=8281cfeb       !
! RSI=8dcdb1e0 RDI=82959be3 CR0=8001003b CR3=00185000 CR4=000026d9 CS=0008       RFLAGS=00000246  !
!-------------------------------------------------------------------------------------------------!
! 8281cfeb: c20400                        ret $0x4                                                 !
! 8281cfee: 8bff                          mov %edi, %edi                                           !
! 8281cff0: 33c0                          xor %eax, %eax                                           !
! 8281cff2: 8b542404                      mov 0x4(%esp), %edx                                      !
! 8281cff6: 66ed                          in %dx, %ax                                              !
! 8281cff8: c20400                        ret $0x4                                                 !
! 8281cffb: 90                            nop                                                      !
!-------------------------------------------------------------------------------------------------!
! Available commands:                                                                              !
! h - show this help screen                                                                        !
! r - dump guest registers                                                                         !
! p - dump guest processes                                                                         !
! m - dump guest kernel modules                                                                    !
! w - dump guest network connections                                                               !
! x addr|$reg [y] [cr3] [char] - dump y dwords starting from addr or register reg                 !
! b addr|$symbol [cr3] - set sw breakpoint @ address addr or at address of $symbol                !
! B addr|$symbol [cr3] - set permanent sw breakpoint @ address addr or at address of $symbol      !
! D id - delete sw breakpoint #id (always check id with BP listing)                               !
! L - list sw breakpoints                                                                          !
! s - single step                                                                                  !
! d [addr] [cr3] - disassemble starting from addr (default rip)                                   !
! c - continue execution                                                                           !
! t n - print backtrace of n stack frames                                                          !
! S addr - lookup symbol associated with address addr                                              !
! n addr - lookup nearest symbol to address addr                                                   !
! i - show info on HyperDbg                                                                         !
!                                                                                                  !
!-------------------------------------------------------------------------------------------------!
! >                                                                                                !
+------------------------------------------------------------------------------[ Made in Italy ]-+
```

```
+--=[pid: 0004; tid: N/A; name: System]=----=[ HyperDbg ]=------------------------------------------+
! RAX=00000058 RBX=00000000 RCX=00002ee0 RDX=00000060 RSP=8292cba4 RBP=8292cbb8 RIP=82c26feb       !
! RSI=8dce71e0 RDI=8292cbe3 CR0=8001003b CR3=00185000 CR4=000426d9 CS=0008       RFLAGS=00000246  !
!-------------------------------------------------------------------------------------------------!
! 82c26feb: c20400                        ret $0x4                                                 !
! 82c26fee: 8bff                          mov %edi, %edi                                           !
! 82c26ff0: 33c0                          xor %eax, %eax                                           !
! 82c26ff2: 8b542404                      mov 0x4(%esp), %edx                                      !
! 82c26ff6: 66ed                          in %dx, %ax                                              !
! 82c26ff8: c20400                        ret $0x4                                                 !
! 82c26ffb: 90                            nop                                                      !
!-------------------------------------------------------------------------------------------------!
!          CR3               PID               Name                                                !
! 000.     0x00185000        0x00000004        System                                              !
! 001.     0x272f0000        0x00000108        smss.exe                                            !
! 002.     0x2239b000        0x0000015c        csrss.exe                                           !
! 003.     0x21ddf000        0x0000018c        wininit.exe                                         !
! 004.     0x23129000        0x00000194        csrss.exe                                           !
! 005.     0x2272d000        0x000001cc        winlogon.exe                                        !
! 006.     0x2247c000        0x000001ec        services.exe                                        !
! 007.     0x22162000        0x000001f4        lsass.exe                                           !
! 008.     0x221e6000        0x000001fc        lsm.exe                                             !
! 009.     0x209ca000        0x00000264        svchost.exe                                         !
! 010.     0x21d3b000        0x000002a4        svchost.exe                                         !
! 011.     0x20381000        0x000002d8        svchost.exe                                         !
! 012.     0x1fe86000        0x00000330        svchost.exe                                         !
! 013.     0x2014b000        0x0000034c        svchost.exe                                         !
! 014.     0x1e451000        0x000003b8        audiodg.exe                                         !
! 015.     0x027d2000        0x000003e8        svchost.exe                                         !
! 016.     0x1df1b000        0x00000424        svchost.exe                                         !
! 017.     0x1c7aa000        0x000004dc        dwm.exe                                             !
! 018.     0x1a8c8000        0x000004e8        explorer.exe                                        !
! 019.     0x195c2000        0x00000534        spoolsv.exe                                         !
! 020.     0x193c4000        0x00000550        svchost.exe                                         !
! 021.     0x1ac37000        0x0000057c        taskhost.exe                                        !
! 022.     0x1a0e3000        0x000005c8        VMwareTray.exe                                      !
! 023.     0x1a3af000        0x00000600        vmtoolsd.exe                                        !
! 024.     0x098c0000        0x000006b8        vmtoolsd.exe                                        !
! 025.     0x10fdb000        0x00000798        svchost.exe                                         !
! 026.     0x0db77000        0x000007a8        svchost.exe                                         !
! 027.     0x0cc21000        0x00000900        wmpnetwk.exe                                        !
! 028.     0x0cda8000        0x00000918        msdtc.exe                                           !
! 029.     0x38385000        0x00000e20        sppsvc.exe                                          !
! 030.     0x37c94000        0x00000e50        svchost.exe                                         !
! 031.     0x194ed000        0x00000b64        portmon.exe                                         !
! 032.     0x2a2e6000        0x00000a8c        sh.exe                                              !
! 033.     0x2d04b000        0x00000fc8        conhost.exe                                         !
!--=[n: next; p: prev; q: quit]=--=[START]=-------------------------------------------------------!
! > p                                                                                              !
+------------------------------------------------------------------------------[ Made in Italy ]-+
```

Figure 3.3: HyperDbg graphical user interface

3.3.1 User interaction

The main mean of interaction between the user and HyperDbg is the keyboard. To allow a user to control HyperDbg, our debugger must interact with the keyboard both in root and non-root mode. Indeed, when in the latter, the user can invoke the debugger by pressing a key (or a combination of keys). Once in root mode, then, HyperDbg reads every keystroke of the user, so that he/she can input his/hers commands.

Luckily, hardware assisted virtualization offers a very convenient mechanism to intercept keystrokes when in non-root mode. Indeed, we can configure the VM-execution control fields to enable exits for each read operation on the I/O ports of the keyboard. From a higher level perspective, this can be configured by HyperDbg (or any other analysis tool) by registering an handler for `IOOperationPort` events that satisfy the condition *port*=`KEYBOARD_PORT && ` *access*=`read`. For each keystroke, the handler checks whether the pressed key corresponds to the configured hot-key (`F12` by default). If so, the debugger opens its graphical interface and waits for further user input. Otherwise, the handler emulates the read operation and returns to the guest. We decided to leverage emulation in this situation, rather than using the single step based approach used for breakpoints, because read operations on I/O ports are trivial to emulate. On the other hand, to interact with the keyboard in root mode, HyperDbg uses a very simple polling-based method that continuously reads from the I/O ports of the keyboard waiting for the user input.

HyperDbg keyboard driver for root mode does not have any limitation with PS/2 keyboards but it cannot handle USB peripherals. This is due to the complexity of USB controllers, much greater than PS/2. Furthermore, while PS/2 controllers are quite stable, given their age, there are many USB version and implementations, that often coexist on the same machine and require custom handling and interaction procedures. The lack of support for USB greatly limits the usability of HyperDbg, since more and more motherboards are produced without PS/2 ports. For this reason, we are investigating solutions to make HyperDbg more usable, even in the presence of USB keyboards.

3.3.2 User interface

As observed from Figure 3.3, the GUI of HyperDbg is quite rough and bare. Nonetheless, its implementation is everything but simple. The main reason behind this is that since HyperDbg operates in root mode, it cannot rely on any high-level function offered by the guest OS to interact with the graphic hardware. This forced us to write our own driver for the video card. The main interaction with graphical hardware happens through video memory: the driver maps the video card memory into virtual memory, and directly writes into this memory area. Such writes will be reflected by the video hardware on the screen attached to it. HyperDbg video driver takes care of locating, through a PCI bus scan, the physical address of the video card and mapping it into its own address space, to make it accessible through a virtual address when in root mode. On top of this driver, we developed a small video library that allows to draw 8x12 pixels fonts on arbitrary coordinates on the screen. HyperDbg uses this library to draw its GUI as soon as it is triggered by the user, or by an event. The driver takes care of saving the content of the video card memory just before writing to it, so that it is possible to restore its original state before entering in the guest.

The main limitation of our "low hanging fruit" approach to handle graphic hardware is that it cannot deal with modern graphic accelerators. Indeed, we basically operate in a VGA standard mode in which RGB values for each pixels are written directly in the graphic card memory (according to the card configuration, either 8, 16, 24, or 32 bits per pixel). When accelerated mode is used, values in such memory do not represent single pixels anymore. Rather, such memory is used for communication between the hardware and a driver that can perform complex tasks more efficiently (e.g., ask the hardware to draw a square on the screen rather than writing each single pixel of the square in the correct memory location). Thus, if the graphic card is in this mode when HyperDbg is loaded, its user interface may not work properly. The best workaround for this problem would be to implement small drivers to handle as many hardware accelerated graphic cards as possible and embed them into our debugger. Given the often closed-source nature of this hardware, however, it would be a daunting task.

An alternative to the graphic interface we recently developed in HyperDbg, that also bypasses the problem of USB keyboards, is to leverage an in-guest user-space program to provide a gdb-like interface to our debugger. Such software communicates via hypercalls (see section 2.2) with the hypervisor, that exposes its API through this guest-to-hypervisor communication interface. This approach, however, suffers from two main problems. First, launching a process in the guest greatly reduces our claims of having a minimum impact on the guest system, thus it could be used only under certain circumstances. Second, events can be handled only *asynchronously*. For example, hitting a breakpoint causes the guest system to completely freeze until the hypervisor performs an entry, including the user-space interface, that is unable to perform any action in response to the event. To overcome this limitation, user-space program uses an approach analogous to gdb `command` directive that allows to define a set of commands to be executed when a breakpoint is hit. By doing so, the user-space interface instructs the hypervisor to trap an event and perform some actions in response to this event. To communicate the results of the operations to the user-space process, the hypervisor uses its memory handling primitives, explained in section 3.2.3, to write results directly into the memory of the process.

Yet another alternative, similar to what Xen [11] does, consists in creating a second virtual machine, parallel to the analyzed guest, in which to run a full operating system. We will call this the *analyzer guest* for clarity. Software running in the analyzer guest could leverage the API of our framework, exposed through the hypercall interface, as in the aforementioned solution, to inspect and trace the analyzed guest. By doing so, analysis procedures run in the analyzer guest would benefit both from our framework and from the facilities of the analyzer guest (e.g., network and graphic libraries). While very promising, this solution would greatly increase the complexity of our framework. For instance, since it is loaded at runtime through late launching, we would need to reorganize the resources of the analyzed guest to allow the boot of the analyzer guest (e.g., get enough memory), and create a mechanism to quickly switch between the analyzed and the analyzer guests. In the current implementation, indeed, our lightweight hypervisor is configured to deal with just one guest. Nonetheless, given the amount advantages that an analyzer guest would introduce, we will inspect this possibility as a future development of our framework.

3.4 Discussion

In this chapter we described an analysis framework that provides an isolated execution environment to run security tools. Once installed, such environment cannot be altered by any malicious program. However, if the system to be analyzed has already been infected, then a malware controlling the machine can still tamper with the framework *before* it is installed, thus making the whole infrastructure completely ineffective. In this situation, a possible solution is to *attest* that the module responsible for loading the analysis infrastructure runs un-tampered, and that the framework itself is not modified by the malware during the installation process. Conqueror by Martignoni *et al.* supports this kind of attestation, with the only requirement of having a trusted host in the same local network [79]. Another possibility is Intel Trusted Execution Technology (TXT) [47] that has been recently introduced on Intel CPUs to, among other things, permit a verifiable and secure launch of an hypervisor. Intel TXT has also been extensively investigated as a support for trusted computing by McCune *et al.* [84].

The effectiveness of our hypervisor framework depends on the impossibility to detect its presence from the guest. There has been much discussion about the transparency of hardware-assisted hypervisors: several researchers believe hypervisors necessarily introduce some discrepancies and suggest a number of detection strategies [40, 2], while others insist this is not always the case, and that similar techniques can be evaded [111]. In our opinion, writing an undetectable hypervisor is extremely challenging, if not unfeasible [42]. For example, timing analysis is very effective in detecting running hypervisors, especially when the analysis is performed by an external entity, with a real perception of time. The reader could argue that malware authors can use similar strategies also to detect our analysis framework. Anyway, we believe it is not realistic to assume that a malicious program can rely on an external entity to perform the timing analysis, and internal time sources can be altered by the hypervisor. Furthermore, the malware would need precise timing information for each infected system, and this seems, at the very least, unlikely. Alternatively, malware might attempt to detect our running hypervisor by trying to install another hypervisor. One approach to contrast such attempts is to let the malware believe that virtualization support is not available at all.

Secondly, it is worth noting that hypervisor detection techniques can check if a VMM is installed on the system, but they cannot easily discern what kind of hypervisor is actually present (at least with timing-based techniques). With the widespread diffusion of virtualization technologies, this kind of detection is often too coarse grained. A malware that hides its malicious behavior on a machine only because it finds a running hypervisor will not be able to infect many of the virtualized environments in use today (e.g., Amazon EC2).

To conclude, we must introduce direct memory access (DMA) attacks [141]. A malware running inside a guest operating system can leverage hardware devices that have DMA capabilities (such a network interface card or a firewire device) to access memory areas of the hypervisor, bypassing memory virtualization. A possible solution to this problem is to leverage Intel VT-d [54]. This support is natively conceived to isolate and restrict device accesses to specific guests but it can also be used to protect the address space of the hypervisor, since it allows to prevent guests from accessing particular memory areas via DMA.

Even with such a support, however, the handling of I/O virtualization is extremely complex and dangerous errors are always possible. As recently discovered [41], indeed, it is possible to exploit misconfigurations of interrupt or DMA remapping to crash the hypervisor (thus freezing the whole physical system) or even to “escape” from the guest and execute code with hypervisor privileges.

The current implementation of our framework needs to be further improved for what concerns DMA attacks, and we are working to include support for VT-d technology, in order to better protect our hypervisor against such attacks.

4 Hypervisor-based malware protection with AccessMiner

The problem of detecting attacks and malicious applications at the host level has been largely studied by both research and industrial communities. The most common solutions are based either on matching static signatures or on using behavioral models to specify allowed or forbidden behaviors. Signatures works well to identify single malware instance but they quickly become ineffective when the attacker adopts obfuscated or polymorphic code [72, 91]. At the same time, most behavior-based detection techniques follow a *program-centric* approach that focuses on modeling the execution of individual programs. These models often lack the context to capture how *generic* benign and malicious programs interact with their environment and with the underlying operating system. As a result, detectors based on program-centric behavioral techniques tend to raise alerts whenever a new program is encountered or an existing program is used in a different way. This typically leads to unacceptably high false positive rates—thus limiting the practical applicability of these approaches.

AccessMiner [67] introduced a novel *system-centric* technique to model the activity of benign programs. The main idea behind AccessMiner approach is that, given enough training, it is possible to identify common patterns in the way benign applications interact with the operating system resources. For instance, while normal programs typically write only to their own directories (and to temporary directories), malware often attempt to tamper with other applications and critical system settings, often residing outside the normal application "scope". As a result, special *access activity models* can be derived by AccessMiner only by looking at the execution of a broad set of benign applications. Therefore, traces of malware execution, often problematic to collect from a coverage and diversity point of view, are not required to train our classifiers.

While experiments showed that AccessMiner system-centric approach was successful in identifying a large amount of diverse malware samples with very few false positives, a number of important points were not addressed in the original paper. In particular, the original approach was designed to be implemented as part of the Windows operating system kernel. However, the threat scenario rapidly changed in the last years, moving from traditional user-space malware toward more sophisticated rootkits techniques. The rise of targeted attacks also pose new challenges that are not fully addressed by current methodologies. For example, by carefully combining a mix of social engineering, zero days exploits for unknown Windows vulnerabilities, and stolen certificates to sign kernel modules, motivated and well-funded attackers can quickly subvert the target OS and remain undetected for long periods of time (as the Stuxnet [125], Duqu [124], and Flame [126] incidents have shown). This being considered, the original implementation of AccessMiner fails to respect one important prerequisite of malware analysis, as identified by Rossow *et al.* [107]: operate at a privilege level higher than the malware.

Since a successful targeted attack could easily tamper with OS-based detection mechanisms, in this chapter we describe a new version of AccessMiner, and how it can be implemented on top of the bare metal hypervisor-based analysis framework we illustrated in chapter 3. This new solution makes the detector much more resilient to sophisticated attacks, but also introduces several technical problems and challenges. First, in order to collect the same information and monitor the system calls issued by each process, a hypervisor has to solve the so-called *semantic gap*. Our framework gracefully tackle this problem, allowing solutions built on top of it to rely on its virtual machine introspection API (see section 3.2). Yet, as we will see, we need to introduce some custom improvements, to make it suitable for AccessMiner requirements. Indeed, even though many solutions exists for this problem, other hypervisor-based detection countermeasures do not scale well to several scenarios (e.g., critical infrastructures), due to their high computational requirements that conflict with the strict timing constraints of the running applications. The challenge here is to use a light-weight approach that does not impact the performance of the system in a prohibitive way.

4.1 System call data collection

In this section, we describe how the set of system call traces used by AccessMiner to build access activity models was collected. Access activity models are built by AccessMiner to generalize how benign programs interact with the operating system, and are later used to identify potentially malicious programs that do not respect a model.

The data collection framework that was built to gather data for access activity models is capable of extracting a rich set of attributes for each event (i.e., system call) of interest. The system consists of a number of software agents, which, once installed on users' machines, automatically *collect*, *anonymize*, and *upload* system call logs to a central data repository, which normalizes the data for further analysis. The software agents can be installed by users on their own machines and are mindful of system load, available disk space, and network connectivity. Furthermore, users can enable and disable the collection agent as they wish.

AccessMiner analysis and training algorithms need several information regarding each system call. Therefore, our sensors were designed to collect the syscall number, its arguments, its result (return code), the issuing process ID, name, and parent process ID. Each log entry is represented by a tuple in the form:

$$\langle \mathit{timestamp}, \mathit{program}, \mathit{pid}, \mathit{ppid}, \mathit{system\ call}, \mathit{args}, \mathit{result} \rangle$$

This data allows to perform analyses within a single process, across multiple executions of the same program, or across multiple programs.

None of the components of the collection framework was implemented at the hypervisor level. This design choice may look a puzzling contradiction of everything we said so far. Instead, it was based on the assumption that users' machines were not compromised at the time of collection, and on the observation that it is much simpler to leverage components that can interact with the OS primitives to perform collection tasks (e.g., pushing data to the repository).

4.1.1 Raw data collection

The software agent that collects data is a real-time component running on each user's machine. This agent consists of a data collector and a data anonymizer. The description in the remainder of this section provides details specific to the Microsoft Windows platform, the system for which the collector was implemented. The data collector is a Microsoft Windows kernel module that traces system call events and annotates them with additional process information. The data anonymizer transforms the collected system call data according to privacy rules and uploads it to the remote, central data repository.

Kernel collector. The main goal of this component is to collect system call and process information *across the entire system*. In order to intercept and log system call information, the kernel data collector hooks the SSDT table [52]. The kernel collector logs information for 79 different system calls in five categories: 25 related to files, 23 related to registries, 25 to processes and threads, one related to networking, and five related to memory sections. Such system calls have been selected to match those used by Anubis [57] that well characterize actions that manipulate the resources of the operating system.

Log anonymizer. To protect the privacy of users, the arguments of various system calls were obfuscated, or simply removed, before sending the log to the data repository. The obfuscation consists of replacing part or the entire sensitive argument value with a randomly-generated value. Every time a value repeats, it is replaced with the same randomly-generated value, so that we can recover correlations between system call arguments. All arguments whose values specify non-system paths (e.g., paths under C:\Documents and Settings are sensitive), all registry keys below the user-root registry key (`HKLM`), and all IP addresses are considered sensitive. Furthermore, the anonymization removes all buffers read, written, sent, or received, thus both providing privacy protection and reducing the communication to the data repository. The data repository indexes the logs by the primary MAC address of each machine.

Impact on performance. The software agent was designed to minimize the overhead on users' activities. The kernel module collects information only for a small subset of the 79 system calls. Logs are saved locally and processed out of band before being sent to the server, when network connectivity is available. Users can turn data collection on and off, based on their needs. Local logs are uploaded to the repository when they reach 10 MB in size and logging is automatically stopped if available disk space drops below the 100 MB threshold. Each 10 MB portion of the system call log is compressed using ZIP compression, for an 95% average reduction in size (from 10 MB to 500 KB).

4.1.2 Data normalization

The purpose of this component is to process raw system call logs and extract fully qualified names of accessed resources as well as the access type. For files and directories, the fully qualified name is the absolute path, while for registry keys, it is the full path from one of the root keys.

To compute fully qualified resource names, the collector tracks, for each process, the set of resources open at any given time, via the corresponding set of OS handles. When a resource (file or registry key) is accessed relative to another resource (either opened by the process or opened by the OS automatically for the process), the collector combines the resource names to obtain a fully qualified name.

Computing the access type (e.g., read, write, or execute) requires tracking the access operations performed on a resource. This is more tricky than expected. When a resource is acquired by a program (e.g, a file is opened), the program specifies a desired level of access. This information, however, is not sufficiently precise. This is because, often, programs open files and registry keys at an access level beyond their needs. For example, a program might open a file with `FULL_ACCESS` (i.e., both read and write access), but afterward, it only reads from the file. Since the collector must gather the actual access type, it tracks every operations on a resource, and only when the resource is released (on `NtClose`), it computes the access type as a union of all operations on the resource.

4.1.3 Experimental data set

Data was collected by the collection framework from ten different Windows machines, each belonging to a different user. The users had different levels of computing expertise and different computer usage patterns. Based on their role, the machines can be classified as follows: two were development systems, one was an office system, one was a production system, four were home PCs, and one was a computer-lab machine.

Overall 114.5 GB of data were collected, consisting of 1.556 billion of system calls, from 362,600 processes and 242 distinct applications.

4.2 System-centric models and detection

Several studies have shown that models based on system call sequences (n-grams) have difficulties in distinguishing normal and malicious behaviors [67, 20]. One of the main problem is that, while n-grams might capture well the execution of individual programs, they poorly generalize to other applications. The reason is that the model is closely tied to the execution(s) of particular applications; we refer to this as a program-centric detection approach.

AccessMiner uses a model that attempts to abstract from individual program runs and that generalizes how benign programs interact with the operating system. For capturing these interactions, we focus on the file system and the registry activities of Microsoft Windows processes. More precisely, we record the files and the registry entries that Windows processes read, write, and execute (in case of files only). We call this an "*access activity model*".

The model is based on a large number of runs of a diverse set of applications, and it combines the observations into a single model that reflects the activities of *all* programs that are observed. For this to work, we leverage the fact that we see "convergence". That is, even when we build a model from a subset of the observed processes, the activity of the remaining processes fits this model very well. Thus, by looking at program activity from a system-centric view—that is, by analyzing how benign programs interact with the OS—we can build a model that captures well the activity of these programs. Of course, this would not be sufficient by itself. To be useful, our model must also be able to identify a reasonably large fraction of malware.

Creating access activity models. Access activity models capture normal (i.e., benign) interactions with the file system and the Windows registry. An access activity model specifies a set of labels for operating system resources. OS resources are directories in the file system and sub-keys in the registry (sub-keys are the equivalent of directories in the file system). For simplicity, in the following we refer to directories and sub-keys as "folders".

A label L is a set of access tokens $\{t_0, t_1, \ldots, t_n\}$. Each token t is a pair $\langle a, op \rangle$. The first component a represents the application that has performed the access, the second component op represents the operation itself (that is, the type of access).

In the current implementation, AccessMiner refers to applications by name. In principle, this could be exploited by a malware process that decides to reuse the name of an existing application (that has certain privileges). As a future work, we plan to replace application names by names that include the full path, the hash of the code that is being executed, or any other mechanism that allows us to determine the identity of the application that a process belongs to. In addition to specific application names, we use the star character ($*$) as a wild card to match any application.

Initial access activity model. An initial access activity model reflects exactly all resource accesses that appear in the system-call traces of all benign processes that we monitored. Note that for this, we merge accesses to resources that are found in different traces and even on different Windows installations. In other words, we build a "virtual" file system and registry that contains the union of the resources accessed in all traces.

Pre-processing. Before proceeding with the generation of the model, the initial model undergoes two pre-processing phases. First, a small set of benign processes that either read or execute files in many folders is removed. This prevents the generation of an excessively loose access activity model.

The second pre-processing step is needed to identify applications that start processes with different names. We consider that two processes with different names belong to the same application when their executables are located in the same directory. Merging processes that have different names but that ultimately belong to the same application is useful to create tighter access activity models.

Both these pre-processing step do not require a particular effort. The set of benign process used by the first step is small, as it mainly includes Windows services and programs, and anti virus software. The number of different applications that belong to these categories is likely small enough so that a manually-created white list could cover them. The second step is also pretty simple, and scales linearly with the number of applications that run processes with different names.

Model generalization. The model is further refined in a last generalization step. This is needed because we clearly cannot assume that the training data contains all possible programs that can be installed on a Windows system, nor do we want to assume that we see all possible resource accesses of the applications that we observed. Also, the initial model does not contain labels for all folders (recall that the access is only recorded for the folder that contains the accessed entity).

The generalization algorithm has four main steps, each of them refining the knowledge contained in the model.

Step 1: identifies folders that are accessed by only one application as belonging to this application and alters labels to reflect this ownership.

Step 2: identifies *container* folders, i.e., folders that are owned by many applications. A typical example of a container folder is the C:\Program Files directory. Since a container holds folders owned by many different applications, its label would deny access to all sub-folders that were not seen during training. As we will see, folders identified as containers are subject to less tight controls in the detection phase.

Step 3: merges access tokens in the label associated with a folder. For example, if multiple applications perform identical operations in a particular folder, we assume that other applications (which we have not seen) are also permitted similar access.

Step 4: finally, the algorithm adds access tokens that were likely missed because training data is not complete. In particular, labels that contains only write access tokens to a folder are extended to include read operations too. The rationale for this step is that an application that can write to resources in a folder can very likely also perform read operations. While it is possible to configure files and directories for write-only access, this is very rare. On the other hand, adding read tokens allows us to avoid false positives in the more frequent case where we have simply not seen (legitimate) read operations in the training data.

Model enforcement and detection. Once an access activity model M is built, it can be deployed in a detector. More precisely, a detector can use M to check processes that attempt to read, write, or execute files in directories or that read or write keys from the registry.

The basic detection algorithm is simple. Assume that an application `proc` attempts to perform operation `op` on resource `r` located in \path\dir. We first find the longest prefix P shared between the path to the resource (i.e., \path\dir) and the folders in the virtual tree stored by M. For example, when the virtual file system tree contains the directory C:\dir\sub\foo and the accessed resource is located in C:\dir\sub\bar, the longest common prefix P would be C:\dir\sub. We then retrieve the label L_P associated with this prefix and check for all access tokens that are related to operation `op` (actually, after generalization, there will be at most one such token, or none). When no token is found, the model raises an alert. When a token is found, its first component is compared with `proc`. When the application names match or when the first component is $*$, the access succeeds. Otherwise, an alert is raised.

The situation is slightly more complicated when the folder that corresponds to the prefix P is marked as *container*. In this case, we have the situation that a process accesses a sub-folder of a container that was not present in the training data. For example, this could be a program installed under C:\Program Files that was not seen during training. In this case, the access is *permitted*, both read and write, to the container, so that the new resource (i.e., the subfolder) can be created without triggering the detector. Moreover, the model is dynamically extended with the full path to the resource, and all new folders receive labels that indicate that application `proc` is its owner. More precisely, we add to each label access tokens in form of $\langle \texttt{proc}, \texttt{op} \rangle$ for all operations. This ensures that from now on, no other process can access these newly "discovered" folders. This makes sense, because it reflects the semantics of a container (which is a folder that stores sub-folders that are only accessed by their respective owners).

Whenever an alert is raised, we have several options. It is possible to simply log the event, deny that particular access, or terminate the offending process.

4.3 Hypervisor framework design

In this section we present the design of the hypervisor based detector that implements the system centric technique presented in the previous section.

Our enforcement model exploits hardware virtualization support available in commodity x86 CPU [92, 4], that we richly discussed in section 2.2. In particular, we built AccessMiner detector on top of the framework we presented in Chapter 3. By doing so, the detector module is tamper-resistant, efficient, and is able to take over the OS operations and verify the policies derived from the AccessMiner system, without requiring any previously installed support.

In the following of this chapter, for simplicity, we use the terms "framework" and "hypervisor" to refer to the analysis framework presented in chapter 3, and "detector" to refer to the AccessMiner module built on top of it.

4.3.1 Threat model

The threat model we adopt considers a very powerful attacker who can operate with kernel-level privileges. On the other side, the attacker does not have physical access to the machine and, therefore, cannot perform any hardware-based attack and he cannot tamper with the hypervisor operations. We assume that our hypervisor starts during the boot process of the machine and it is the most privileged hypervisor on the system.

Is it also possible to leverage *late-launching* capabilities of the framework to load it after the boot. For this to be feasible, however, we have to relax our threat model a little. Indeed, we must assume that either there is no malware on the machine *before* we launch AccessMiner or that we leverage an integrity checking technique to ensure that the hypervisor is not altered at load time [79, 84]. Despite this requirement, a hot-bootable detector module can be quite useful in scenarios where it is not possible to restart the machine (e.g., when it provides some critical

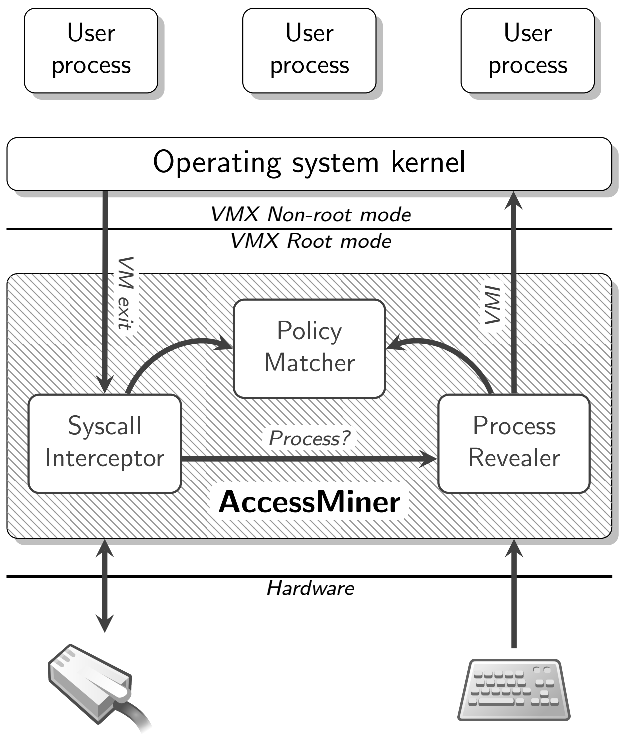


Figure 4.1: Hypervisor Architecture.

service).

4.3.2 Hypervisor architecture

The detector system is composed by three components: a *system call interceptor*, a *policy matcher*, and a *process revealer*. The outputs of all the components are combined together to check the policies derived by AccessMiner system. In Figure 4.1, we depict a scheme of the overall architecture, hiding details of the underlying framework.

Detection & management mode. Our detector supports two different operation modes: Detection and Management mode. The first is responsible for the detection of attacks against the system, while the latter is used to receive configuration instructions from an external machine, using a dedicated network protocol. To this end, we leveraged our framework capability to intercept a particular sequence of keystrokes, but we had to implement a custom network driver from scratch.

Switching from Detection to Management mode is done by pressing a keystroke sequence. The hypervisor can easily intercept every keystroke by registering for I/O events of the keyboard (like shown in section 3.3) and, if the sequence is correct, it runs a communication procedure to receive and update its configuration.

The communication protocol we implemented supports two types of commands: one to receive policies configuration and one to load the new set of policies and reset the state of the detector. Both commands are identified by a special identification number. After setting up the configuration, the hypervisor automatically switches from Management to Detection mode and it continues to perform its real-time detection task. When running in detection mode, the detector is configured to intercept and handle two privileged instructions: write operations on Control Registers and alterations to Machine Status Registers (MSR) [55]. This allows us to detect context switches among processes and to verify the trusted path of our system call hooking mechanism. More details about the monitoring task are provided in the next section.

System call tracer. The core of the detector is represented by the System Call Tracer component. Its goal is to intercept the operations performed by the OS, in terms of system call type, parameters and return values. All these information will be used by the Policy Matcher, to verify the right permissions on a certain resources on behalf of the process. There are two main requirements for this component: (R1) The interception mechanism must provide a trusted path between the invocation of the system call and its own termination. In particular the system needs to provide a secure hooking mechanism for intercepting the invocation and termination of OS operation. (R2) The overhead of the interception mechanism must be kept as low as possible.

In order to retrieve all system call information, we need to monitor the invocation of the operation along with its termination. Whenever a system call is issued by a process, a `sysenter` instruction is invoked. The `sysenter` instruction refers to the `SYSENTER_EIP` MSR, that contains the address of the system call handler. Instead of relying on breakpoints offered by our framework, we directly overwrite the `SYSENTER_EIP` MSR so that it points to a `vmcall` instruction. By doing so, the hypervisor intercepts every executed system call and notifies the event to the detector, that accurately parse the parameters according to their type. Note that the hypervisor has the ability to intercept *every* attempt that the guest makes to read *or* write such MSR. As we described in section 2.2, indeed, the hypervisor can configure vm exit control fields so that an access to a defined MSR triggers an exit. By doing so for the `SYSENTER_EIP` MSR, our hypervisor is able to efficiently protect the system call interception mechanism (requirement R1), both from unwanted reads *and* write operations. Note that there are other ways to intercept system calls, e.g., the transparent breakpoint-based event originally offered by the framework (see section 3.2.2). However, given our ability to easily mask the modifications to MSRs, we decided to relax a little the “non modification” standard of the framework and adopt the solution we just illustrated, that is more efficient for our needs.

Before passing the information to the Policy Matcher, the tracer also needs to check whether the operation is successful or not and to collect its return value. For this purpose, the detector is able to intercept a `sysexit` instruction by replacing it with a `vmcall`. Any attempt to re-write the VMX instruction is prevented by the hypervisor through a memory page protection mechanism (requirement R1). Once again, this could have been implemented through transparent breakpoints offered by our framework. However, write protecting that page is extremely more efficient than removing its execution permissions (as done in transparent breakpoints).

To verify the trusted path of the system call, our detector also implements a simple automaton that checks the correctness of the system call execution flow. Every time a `sysenter` is intercepted an opening bracket "(" transition is triggered to indicate which system call was invoked. Every time a `sysexit` is intercepted, the hypervisor verifies that the event was expected, given the invoked system call. It performs this step repeatedly until it sees the subsequent ")", corresponding to the end of the system call request. Any unknown state is reported as a system anomaly. If the operation succeeded, the System Call Tracer invokes the Policy Matcher component and provides all the information on the system call type, parameters, and return value.

Since the hypervisor is intercepting a high number of system calls, the hooking mechanism is a critical component from a performance point of view. Consequently, to improve performances, we devise two modifications to our original implementation. First, the system allocates a protected memory page that contains a short control code and some data about the monitored system calls—such as the system call types and the memory handler code address. Based on the system call type, the code decides whether to invoke an hypercall to switch to monitor mode or to leave the control-flow to the default system call handler. By using this technique we are able to exclude the non-monitored system calls and reduce the overhead of the whole hypervisor system (requirement R2). More details about performance evaluation are reported in the section 4.5.

Another relevant source of overhead is related to possible multiple repetitions of the same system call from the same process. For example, during a file copy operation, the same read and write operations are repeated multiple times, according to the size of disk blocks and of copied file. Since there is no reason to check the permissions for each operation, our system is designed to verify only the first occurrence of the operation and run other operations natively. The overhead caused by the repetition of these operations is thus avoided.

This is implemented by introducing a small cache that contains an checksum based on the system call number, its parameters, and the value of the `CR3` register of the process who is performing the operation. Every time the system discovers a new operation, we insert it into the cache and when the operation is not likely to be repeated (e.g., the corresponding process terminates, or the file is closed), we flush the cache entry related to that operation. In this way we only check the first operation and we skip possible repetitions (requirement R2). We report a measurement of the effectiveness of our cache in section 4.5.

Process revealer. The goal of this component is twofold: First, it extracts and provides the name of the process that is performing the actual operation (i.e., a system call) by leveraging introspection functions of the framework (section 3.2.3) and, second, it caches these information to reduce the system overhead. The component keeps a cache that allows to look up the name of the process given a certain `CR3` value. The cache is updated every time a process is created or destroyed, by properly intercepting and analyzing process-related system calls.

Policy checker. The goal of this component is to check AccessMiner policies and generate an alert in case some of them are violated.

We recognize two main phases for the Policies Checker task: Initialization and Detection phase. The initialization phase is responsible to create the memory structures that will be used for the detection phase. In particular, to check the filesystem and registry policies, we adopt an hash table memory structure where the name of each resource is used as key and the name of process with its own permissions on that resource is stored as value. During the initialization phase, the detector receives the signatures using the ad-hoc network communication protocol we briefly mentioned above. Then, whenever a signature is loaded, the full-pathname of the corresponding resource is extracted and inserted in the memory structure as a key of the hash table. The list of the processes that can get access to the resource along with their own access permission are inserted as elements of such a key.

Another important memory structures used by the policy matcher is the *file/registry handles* structure. Since most of the filesystem and registry system calls operate on handles, while our policy system works with full-pathname resources, the system needs to keep the association between an handle number and the resource full pathname. For this reason, we use a dynamic memory structure that tracks this association. During the monitoring of the system, every time a resource is created or opened, the system retrieves the handle associated to the resource full pathname and it registers it in the structure. Afterwards, when a system call operates on the same handle, the corresponding object is retrieved from the handle structure. Every time an handle is closed, the system removes it from the handles memory structure.

To protect the policy information loaded during the initialization phase, the network driver that receives commands is only enabled when the hypervisor is in Management mode—in our prototype, this is triggered by using a special keystroke sequence. On the other hand, to protect the policies from network attacks, a signature scheme between the hypervisor and the management console is provided. In this way, we can assure that no one is able to tamper the configuration information of the detector, according to our thread model.

During the detection phase, the System Call Tracer invokes the Policy Checker with the relevant system call information. At this point, the Policy Checker, by using the resources full-pathname as a key of the hash table, retrieves the list of the processes along their permissions. It also queries the process Revealer component in order to retrieve the processes name that acts as a subject of the operation. Once all the information are obtained, it scans the list of the processes to search the process name. If the process is not allowed to perform the operation, the Policy Checker raise an alert and blocks the operation. Otherwise, it permits the operation and then returns to non-root mode.

4.4 Detection results

We report the evaluation of the effectiveness of using AccessMiner access activity models to detect malware, conducted on the same machines that were used for the collection. Each machine in turn was protected with the model built with the data gathered from the other 9 machines, and then tested with slightly less than 8,000 malware samples. Since the detection mechanism of AccessMiner has not changed in our hypervisor based implementation, the detection results are unchanged with respect to the original implementation and evaluation. Nonetheless, we report them hereby for completeness.

File system access activity model. On average, the file system access activity model contained about 100 labels with tokens that restrict read access to about 70 directories, write access to about 80 directories, and execute access to about 30 directories.

A first round of experiments with this model reported a very good detection rate, higher than 90% but with an annoying false positive (FP) rate of 14.8%. Different experiments led to the observation that using a *write-only* detection approach greatly reduced the FP rate (4%) without impacting too much on the detection rate (89.5%). Further investigations resulted in the discovery that the remaining FPs were to be ascribed to small errors in the model, and to software updates, during which an application that was never observed to write-access its own directory suddenly tried to, due to its update process. To prevent this issue, we now automatically give write permission to each application on their own directories. Once corrected, the new model allowed us to obtain a 0% FP rate.

Registry access activity model. In our experiments, the registry access activity model contained about 3,000 labels, significantly more than the file-system model. In particular, the labels contained tokens that restrict read access to about 1,600 keys and write access to about 2,800 keys (*execute* is not defined for registry keys).

This model on its own gets worse result than the file system model: 56% detection rate with a 6.6% FP rate. Slightly loosening the model allowed to obtain a 0% FP rate, at the price of a marginally lower detection rate of 48.6%. In particular, all the false positives of the registry model were due to write accesses to a particular registry sub-tree (`HKEY_USERS\Software\Microsoft`), so we allowed write access to this sub-tree and got the results reported above. Despite this being an important part of the registry, the model does not lose much efficacy. Furthermore, as we will see, the file system model alone is responsible for the most part of the detection and, thus, this loosening does not affect the detection of particular kinds of malware. Finally, note that, given enough time for the training, the number of FPs could be reduced even without allowing write access to this part of the registry.

Full access activity model. For the final experiment, we combined those improved file system and registry models that yielded zero false positives. The combined detection rate improves compared to the file system model alone, but only slightly (between 1% and 2% for all of the ten runs). The average detection improved from 89.5% to 91% (of course, with no false positives).

4.4.1 Discussion

When focusing on write operations only, the access activity model achieves a good detection rate (more than 90%) with a very low false positive rate. The false positive rate even drops to zero with minor manual adjustments that compensate for deficiencies in the training data, while still retaining its detection capabilities. This suggests that a system-centric approach is suitable for distinguishing between benign and malicious activity, and it handles well even applications not seen previously. This is because most benign applications are written to be good operating system "citizens" that access and manage resources (files and registry entries) in the way that they are supposed to.

Malicious programs frequently violate good behavior, often because their goals inevitably necessitate tampering with system binaries, application programs, and registry settings. Of course, we cannot expect to detect all possible types of malicious activity. In particular, our detection approach will fail to identify malware programs that ignore other applications and the OS (e.g., the malware does not attempt to hide its presence or to gain control of the OS) and that carry out malicious operations only over the network. For these types of malicious code, it will be necessary to include also network-related policies.

4.5 Performance results

In this section we report a set of micro and macro benchmarks we used to demonstrate the efficiency of our system. In our experiments we run the Passmark Performance Test suite [120] in four different test environments: on a physical machine (PM), inside a guest VMware virtual machine (VM), on physical machine

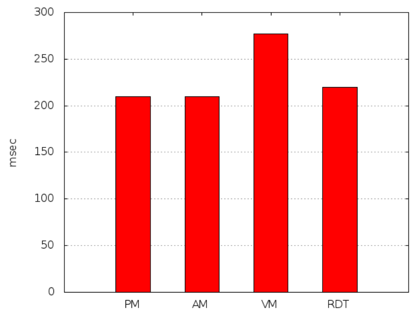


Figure 4.2: Memory Read Operation.

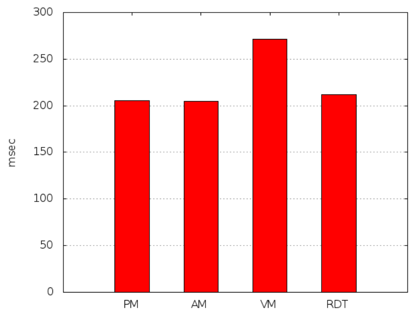


Figure 4.3: Memory Write Operation.

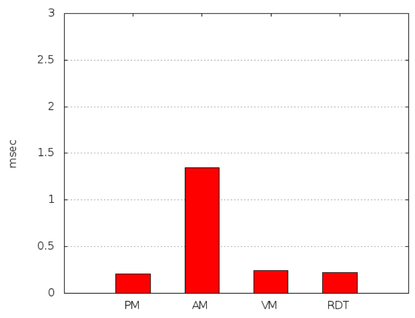


Figure 4.4: Disk Read Operation.

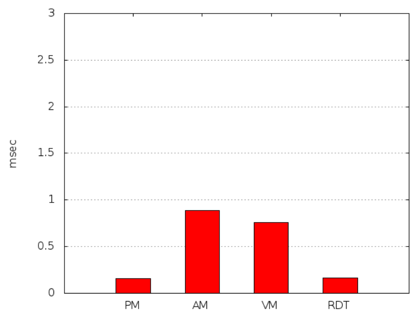


Figure 4.5: Disk Write Operation.

with AccessMiner (AM), and on physical machine running the Hypersight real-time rootkit detector (RTD) [100]. Hypersight is a hardware-supported virtual machine monitor that starts at boot time and intercepts several types of suspicious actions applied to critical memory structures such as attempts to modify page tables, read-only kernel modules, and GDT and IDT tables. All the experiments are performed on an Intel Core i7 2.67 GHz with 3 GB of memory running Windows XP (32-bit).

4.5.1 Macro-benchmark

We measured and compared the overhead introduced by AccessMiner on different workloads by using four of the PassMark performance tests: read and write memory operations and sequential disk read and write operations. To perform these tests, we loaded AccessMiner with 3824 policy rules: 173 related to the filesystem and 3651 related to the Windows registry.

The final values were obtained by taking the average of ten repetitions for each benchmark. Figure 4.2 and Figure 4.3 show the results of the memory tests. In these cases, we used the system to perform a sequential read or write operation of 1GB of memory with a block size ranging from 1024 bytes up to 512 MB. As we can see in the graph, the higher overhead is encountered in VMware, mainly due to its memory virtualization.

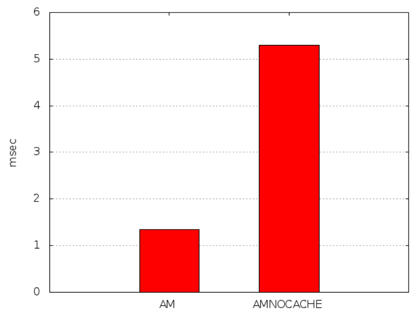


Figure 4.6: Cache/NoCache Disk Read Operation.

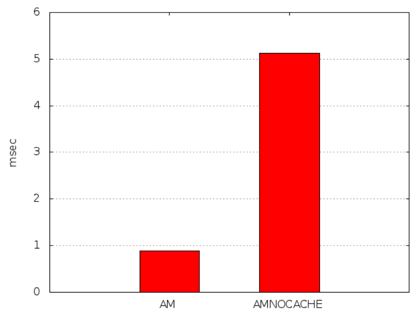


Figure 4.7: Cache/NoCache Disk Write Operation.

Memory Operations	**PM**	**AM**	**VM**	**RTD**
Read (ms)	209.8916	209.9269	277.2247	220.0006
Write (ms)	205.5485	205.1042	271.5995	212.1548
Overhead Read	NA	100.0 %	132.1 %	104.8 %
Overhead Write	NA	99.9 %	132.1 %	103.2 %

Table 4.1: Overhead of Memory Operations.

More interesting performance results are reported in Figure 4.4 and Figure 4.5. For this test, we performed sequential read and write *disk* operations for 1 minute. We use the NTFS filesystem with a block size of 8192 bytes. As we can see in the graphs, the overhead of our system is similar to the one observed in a VMware virtual machine, when dealing with write operations. Read operations in VMware are much faster than writes, because of aggressive caching techniques that it adopts for its virtual disks. Hypersight overhead is the same of the physical machine, since it does not intercept any operation on the disk. The overhead for our system is due to the high number of filesystem and registry system calls performed by the benchmark program. However, in these scenarios with multiple repetitive operations our caching mechanism is able to reduce the overhead of almost 80%, as reported in Figure 4.6 and Figure 4.7.

During the PassMark disk test we counted 11.000 `NtRead/NtWrite` system calls related to the filesystem operations and other 5430 system calls related to the registry operations. The impact of the disk operation was largely covered by our caching system, leaving the registry responsible for most of the overhead.

In table 4.1 we report the overhead of memory operations for the four environments test: PM, AM, VM, and RTD system. In table 4.2 we report the overhead of the disk operations.

To conclude the macro benchmarks, we also performed a worst-case experiment, in which we measured the overhead introduced by AccessMiner during a source code compilation routine. The target of the compilation was a middle-sized C program, composed of almost 100,000 lines of code. Results of this last

Disk Operations	**PM**	**AM**	**VM**	**RTD**
Read (msec.)	0.2070	1.3470	0.2460	0.2240
Write (msec.)	0.1580	0.8900	0.7590	0.1640
Overhead Read	NA	650.7%	118.8%	108.2%
Overhead Write	NA	563.3 %	480.4 %	103.80 %

Table 4.2: Overhead of Disk Operations.

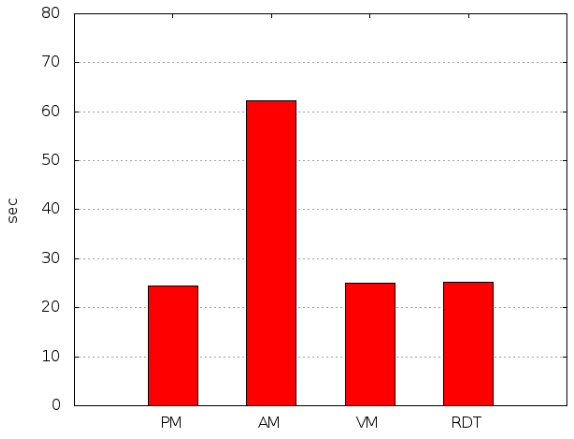


Figure 4.8: GCC evaluation.

benchmark are reported in Figure 4.8. In this case, since the I/O operations were spread on hundreds of different files, our caching mechanism was less effective in mitigating the disk overhead. This resulted in an average AccessMiner overhead, with respect to the physical machine baseline, of around 2.5x.

To conclude, the performances of AccessMiner greatly depend on the type of application. However, the system normally introduce an overhead that is comparable with the one observed in a traditional virtual machine environment.

4.5.2 Micro-benchmarks

To have a more fine-grained view of the delay introduced by our system, we measured the overhead introduced by triggering a system call on a particular resource. We started by measuring the time needed to perform a context switch between a VM exit and a VM entry (without checking any policy), taking an average over 20 repetitions. The operation took 1216 clock cycles, corresponding to around 0.45 microseconds. The second operation that we measured was the entire syscall monitoring mechanism. In this case, the time needed to intercept a single system call is, in average, 1,241,739 clock cycles, or about 0.47ms. These results show that most of the overhead introduced by our system is due to the policy validation mechanism, while the context switch along with the monitoring mechanism do not impact the system in a relevant way.

5 Automatically finding vulnerabilities in OS X kernel extensions

Kernel security is of vital importance for a system. A vulnerability, in any of its parts, may compromise the security model of the whole system. Unprivileged users that find such vulnerabilities can easily crash the attacked system, or obtain administration privileges. Unfortunately, kernels are an evermore attractive target for attackers, and the number of kernel vulnerabilities is rising at an alarming rate [95]. Looking for vulnerabilities in kernel-level code is a daunting task, because of its many intricacies. Indeed, modern kernels are extremely complex and have many subsystems, possibly developed by third parties. Often, such components are not as secure as the kernel, because of the lack of testing, and their typical closed-source nature [96, 97, 98, 99]. Furthermore, kernels have countless entry points for user data. System calls, file systems, and network connections, among others, allow user-fed data to reach important code paths in the kernel. If a bug is found in such paths, it can lead to vulnerabilities that compromise the entire system security.

In this chapter, we present the design and implementation of LynxFuzzer, a framework to automatically find vulnerabilities in *kernel extensions* (kexts), i.e., dynamically loadable components of the Mac OS X kernel. Kexts are built adhering to the *IOKit framework* [6], the official toolkit for creating OS X kernel extensions (also used in the iOS environment).

As its name suggests, LynxFuzzer is based on fuzzing, a technique that is widely adopted for finding security vulnerabilities in software, but seldom applied to the testing of kernel components [58]. We briefly recall that fuzzing consists in applying random mutations to well-formed inputs of a program, and observing the resulting values. There are three different approaches to fuzzing: *whitebox* fuzzing [44], when the source code and other information are available, *blackbox* fuzzing [86, 87], when the target is fuzzed in a completely blind fashion, and *hybrid* fuzzing [49, 68], that leverages a combination of dynamic and static analyses, to fuzz more *"smartly"* than pure blackbox approaches.

This last one is the approach we are interested in, because the *IOKit framework requires* kernel extensions to specify strict constraints on the input values that can be received from user-space [6, 116], thus making blackbox approaches unfeasible: most inputs would be discarded. Thus, LynxFuzzer uses a hybrid fuzzing approach and adopts dynamic test-case generation. This means that it *automatically* extracts information from the target extensions and uses them to *dynamically* adapt its input generation techniques. More precisely, we implemented three different fuzzing engines inside LynxFuzzer: a simple *generation* engine that produces pseudo-random inputs that only respect kext-defined constraints, a *mutation* engine that builds input data from previously *sniffed* valid inputs, and an *evolution-based* engine that leverages evolutionary algorithms [7] and dynamic kernel-level code-coverage analysis.

The implementation of such a strategy posed some challenges, since extracting information from kernel-level components without modifying the components themselves is, practically speaking, not feasible. On the other hand, any kernel code alteration could void testing results. To overcome this problem, we included a hypervisor component inside LynxFuzzer, and use it to inspect a running OS X kernel and intercept any invocation of its extensions, in a completely transparent fashion. We decided to build LynxFuzzer hypervisor component on top of the framework we illustrated in chapter 3 mainly because of the many difficulties of running Mac OS X inside commonly available virtualization software. OS X, indeed, proved to be quite "stubborn" towards virtualization. For example, its installation under an extremely widespread virtualization environment, i.e., VMware [136], requires both a customized OS X disk image and an obscure patch to VMware binaries. On the other hand, we did not experience any particular problem in dynamically loading our hypervisor framework and taking control of an OS X system, proving once again its usefulness. Furthermore, Apple is well-known for its custom hardware (e.g., the Magic Mouse), which is not often emulated by common hypervisors. This shortcoming would prevent LynxFuzzer from testing some of the drivers in OS X. As we illustrated in chapter 3, our framework does not suffer from this problem. Finally, consider how important transparency is for a delicate activity such as the fuzzing of kernel components. We have not experienced this in our experiments, but the OS X kernel could decide to alter its own behavior when it detects the presence of a virtualization environment, maybe refusing to load some components. This would prevent us from fuzzing those components or it may invalidate our results, in case some components behave differently from how they would on real hardware. Using our hypervisor analysis framework, we can ensure as much transparency as possible, eliminating such side effects.

LynxFuzzer has been implemented and tested on Mac OS X 10.8.2 (Mountain Lion), and provided unexpected results, thus proving its usefulness and effectiveness. More precisely, we individuated bugs in 6 of the 17 kexts we analyzed. Once exploited, such bugs can lead unprivileged users to easily crash the whole system or may also allow to mount privilege-escalation attacks [103].

Finally, some last words on the choice of OS X as a test-bed. First, Windows and Linux user-to-kernel interfaces have been deeply analyzed and many tools exist for their testing and verification. On the other hand, user-to-kernel communication approach used by kernel extensions, named *DeviceInterface* [6], is relatively young and has not been extensively analyzed before. Furthermore, the number of OS X adoptions is growing at a continuously rising pace and this will soon attract the attention of more and more cyber-criminals. Finally, note that most of the components of LynxFuzzer can be almost effortlessly adapted to work with iOS kexts, thus opening a whole new scenario of testing on mobile devices. The only component that should be rewritten from scratch is the hypervisor, that could no longer be built on top of our framework, but would require an ARM based hypervisor. As a future development of LynxFuzzer, we plan to implement such component and use it to test iOS kernel extensions as well.

5.1 Preliminary notions

In this section we will give an overview of the main technologies that are basic for the understanding of LynxFuzzer: evolutionary algorithms and the IOKit framework.

5.1.1 Evolutionary algorithms

One of the fuzzing engines that we implemented in LynxFuzzer is based on evolutionary algorithms [7]. Such algorithms are inspired to the natural selection process of biological evolution. A common evolutionary algorithm starts from an *initial population*, a set of elements that are gradually discarded and merged, in an effort to keep only the features of its best elements. To determine which elements are the best, and will contribute to the next generation, the algorithm uses a *fitness function*, that reflects the survival probability of an element. Then, the algorithm iteratively *selects* a number of elements, picking with higher probability elements with a better fitness value. Selected elements are then merged together to build new ones, that share features of both their parents, during the *crossing-over* phase. There are many *cross-over* techniques, but they typically consists in splitting parents in two and concatenate one portion of each one to create the *child* element. Furthermore, elements can undergo *mutations*, that randomly alter their features.

Ideally, each iteration gets a step closer to an optimal solution for the original problem, but there is no *a priori* guarantee that the algorithm will eventually find such solution.

```
const IOExternalMethodDispatch
UserClient::sMethods[kNumberOfMethods] = {
{
 /* kMyScalarIStructIMethod */
 (IOExternalMethodAction) &UserClient::sScalarIStructI,
 /* One scalar input value */
 1,
 /* The size of the input struct */
 sizeof(MySampleStruct),
 /* No scalar output values */
 0,
 /* No struct output value */
 0
},
/* ...  */
}
```

Figure 5.1: A sample dispatch table.

5.1.2 IOKit fundamentals

Mac OS X is an operating system developed by Apple and currently adopted on Apple computers [116]. While most of the concepts behind OS X are shared with other operating systems, some are less well-known and need explanation before proceeding further.

OS X kernel is called *XNU*. It is based mainly on Mach [1], FreeBSD [132] and previous Apple products and prototypes. Despite the fact that Mach uses a microkernel architecture, XNU is a *macrokernel* and it is possible to *extend* its original functionalities by dynamically loading new components into the kernel.

Among its many components, XNU contains one that is particularly relevant for our work: the IOKit. IOKit is a *"a collection of system frameworks, libraries, tools, and other resources for creating device drivers in OS X"* [6]. It offers an object oriented environment (restricted C++) and a number of abstractions that make the (commonly painful) task of writing kernel components much more easy and "user space friendly". The IOKit framework contains many abstractions and components, and we describe those relevant to our work in the following of this section.

Kernel extension. A device driver produced through the IOKit framework is called "*Kernel Extension*" or, for short, *kext* [116]. Since the terms are often synonymous, we will interchangeably use "*kext*", "*module*" and "*driver*" to refer to a dynamically loadable component of the XNU kernel.

Kexts are bundles containing a number of resources needed to properly validate and load a module into the kernel such as the *Info.plist* file, which contains many fundamental fields such as the *CFBundleExecutable*, that points to the driver executable, and the *CFBundleIdentifier*, which contains the driver unique identifier, version number, and the set of libraries on which the driver depends.

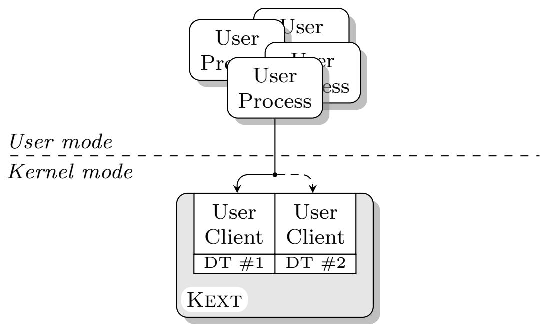


Figure 5.2: Invoking a kext's method.

User to kernel communication. A fundamental component of every OS is the communication between user- and kernel-space components. Windows and Linux offer such functionality through system calls and special virtual files (e.g., `/dev/urandom`). IOKit supports both techniques, but also adds a novel and much more complex mechanism, named *DeviceInterface* [70, 6].

To leverage this mechanism, kexts define a set of methods that can be invoked by user-space programs. Such methods have limitations in terms of number and type of data that can be received by and returned to the caller. The list of these methods and constraints on their parameters are stored into a structure of utmost importance (both for IOKit and for our fuzzing framework): the *dispatch table*.

As can be seen in the Figure 5.1, the dispatch table is an array of structures, each one containing a function pointer, allowed input and output values, as well as the number and size of the values that can be accepted (or that are returned) by the method. If the size of an input structure is not known *a priori*, the check can be demanded to the receiving method, by specifying a size of `0xffffffff` in the table.

It is possible to have, for any driver, more than one dispatch table. IOKit allows kexts to offer multiple interfaces to user-space programs at the same time. Each kext must also define one custom subclass of *UserClient* for each interface. Instances of these subclasses are then loaded in the kernel memory along with the kext (Figure 5.2). Every *UserClient* object contains the dispatch table corresponding to the interface it offers.

User-space programs can invoke kexts methods via `IoConnectCallMethod()`, if the methods are specified in one of the kext dispatch tables. To be able to do so, however, the program must first find the specific instance it wants to talk to. To illustrate how this happens, we must first introduce one more IOKit abstraction, the *IOService*. Each IOKit device driver is an object that inherits from the IOService class and a kext may contain different IOService objects at the same time. Let us consider an example. There are typically many USB devices connected to a computer and each one needs its own driver. Such drivers are all contained in the `IOUSBFamily` kext, and each of them is a specific IOService subclass (*service*, for short) for a device. When an user-space process wants

```
kern_return_t IOConnectCallMethod(
  mach_port_t connection,
  uint32_t selector,
  const uint64_t *input,
  uint32_t inputCnt,
  const void *inputStruct,
  size_t inputStructCnt,
  uint64_t *output,
  uint32_t *outputCnt,
  void *outputStruct,
  size_t *outputStructCntP
);

IOReturn IOUserClient::externalMethod {
  uint_32t selector,
  IOExternalMethodArguments *args,
  IOExternalMethodDispath *dispatch,
  OSObject *target,
  void *reference
}
```

Figure 5.3: Prototype of `IoConnectCallMethod()` and `externalMethod()`.

to communicate with one device, as we mentioned above, it establish a mach connection [70] to IOKit and searches for the specific service associated to the device. This process is called *Device Matching* [6].

Once the communication channel is established and the service is found, the user-space program uses `IoConnectCallMethod()` (whose prototype is reported in Figure 5.3) to invoke the desired method. This transfers the control to the IOKit framework that, before actually executing the target method, performs a set of operations. At first, it retrieves the dispatch table entry from the *User-Client* object. The entry is then passed to the `externalMethod()` function (also reported in Figure 5.3), along with other parameters that allow to perform the actual invocation of the kext method. This, however, happens *only* if the parameters correspond to what is specified by the dispatch table entry, otherwise the invocation is blocked.

The whole IOKit input control mechanism offers an additional protection layer, with respect to common mechanisms such as `ioctl`, because checks are performed before parameters are even moved from user- to kernel-space. Obviously, all these constraints make the fuzzing process quite complex. It is almost useless to just fuzz functions of a kext with completely random parameter sizes, as most invocations would be discarded by checks performed by IOKit. As we will see in the next section, one of the key aspects of our fuzzer is its ability *automatically* extract constraints on parameters from the target and *dynamically* adapt its fuzzing techniques to get better efficiency.

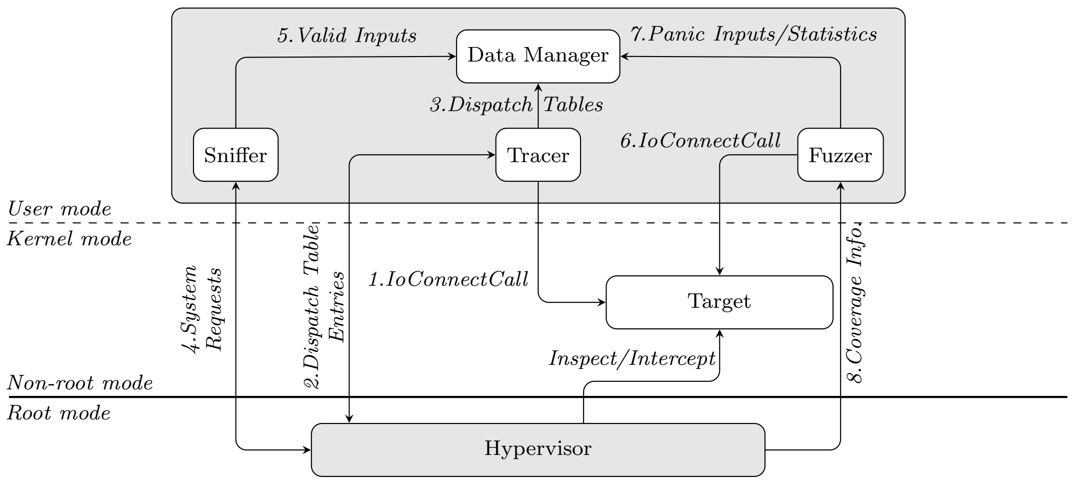


Figure 5.4: Architecture of LynxFuzzer and interactions between its components (gray areas).

5.2 LynxFuzzer

This section describes in detail the design and implementation of LynxFuzzer, the infrastructure we have devised to perform automatic fuzzing of Mac OS X kernel extensions. The purpose of our fuzzer is to trigger bugs in kext code that can be reached by user-space programs, by means of the IOKit framework. Of course, there may be other bugs that the execution flow cannot reach anyhow from user-space, but we give much more importance to the first category, as it poses security threats. A bug triggerable by user-space, indeed, may allow un-privileged users to crash the system, or even obtain arbitrary kernel-level code execution, which typically leads to privilege escalation attacks [103]. Furthermore, we decided to focus our efforts on the *DeviceInterface* boundary-crossing mechanism, since it became the *de facto* user-to-kernel communication standard for OS X kernel extensions.

```
UserClient::sScalarIStructI(
                          uint64_t *ScalarI,
                          uint32_t inputCnt,
                          MySampleStruct *StructI,
                          size_t inputStructCnt
                        );
```

Figure 5.5: Sample kext method.

As it should be clear from what we described in section 5.1.2, there are many constraints that must be considered when invoking a kext method and such constraints are specific to each kext. For example, consider the method in Figure 5.5 (whose constraints are declared in the dispatch table entry in Figure 5.1). Without knowledge of its dispatch table entry, the fuzzing space of this method input parameters has a size of $16 * 2^{64}$: 16 possible values for the scalar vector[1] and 2^{64} for the input structure size only, without considering its contents. On the other hand, knowing constraints contained in the dispatch table allows us to reduce the possible fuzzing space to what is actually accepted by the kext, thus gaining much more efficiency. Thus, we designed LynxFuzzer so that it can extract these information in a completely automatic fashion and fuzz them autonomously. Furthermore, the automation of our fuzzing infrastructure is not limited to this. Indeed, we are able to extract valid input vectors used in non-artificial interactions between user- and kernel-level components. Such inputs are used as a basis to elaborate new inputs aimed at improving the fuzzing strategy.

An overview of LynxFuzzer architecture is depicted in Figure 5.4. Our framework has two main components: one that resides in user-space and consists of 4 sub-components, and one built on top of our hypervisor analysis framework. Figure 5.4 also reports the main interactions between LynxFuzzer internal components. The *tracer* interacts with the hypervisor to identify where the dispatch tables of the target are (1, 2). Once discovered, the hypervisor retrieves them from the kernel memory and sends them back to the tracer that stores them into the *data manager* for later use (2, 3). The sniffer uses such information to *intercept* non-artificial `IoConnectCall()` invocations and gather a set of valid inputs (4, 5). Finally, the fuzzer components starts invoking the target methods with custom parameters (6), waiting for an eventual panic (7). Depending on the selected fuzzing engine, the fuzzer may use valid inputs precedently stored in the data manager to generate new inputs or leverage coverage information (8).

As we mentioned at the beginning of this chapter, the choice of leveraging our hardware-assisted analysis framework was dictated, among other things, by the principle that a fuzzing infrastructure should induce as less changes as possible in its target, i.e., the OS X kernel and its extensions. Any change in the fuzzing target, indeed, could lead to errors and side-effects wrongly identified as produced by the fuzzing strategy (i.e., false positives). For this reason, every functionality that would normally require an interference with the kernel (e.g., intercepting a kernel function) has been implemented at the hypervisor level. Components that do not require to *directly* interact with the kernel are implemented in user-space and communicate with the hypervisor by means of hypercalls.

5.2.1 Tracer

The tracer is the first component of LynxFuzzer to be executed, as its task is finding out *what to fuzz*, i.e., it must identify which of the target methods can be invoked. Intuitively, recalling what we illustrated in section 5.1.2, this information is contained into the dispatch tables of the target kext. Being able to

[1]This is a limit imposed by IOKit.

locate the dispatch table of a kext, however, is not easy, since IOKit uses a number of abstraction layers to hide such information to user-space programs. We devised a solution by observing that, whenever an user-space program invokes `IoConnectCallMethod()`, the IOKit will invoke its `externalMethod()` function (as we illustrated in section 5.1.2). Thus, we proceed as follows. LynxFuzzer hypervisor sets a breakpoint on the `externalMethod()` function and intercepts whenever it is executed (see section 5.2.4 for details on how we implemented breakpoints from the hypervisor). Once the trap is set, the tracer issues a request to the target kext with a `selector` value of 0 and a set of parameters that are likely to be discarded by IOKit checks (we do not want to accidentally trigger a crash during the first phases). When the hypervisor intercepts the resulting `externalMethod()` execution, it extracts the base of the dispatch table, uses it to dump the whole table as described above, and eventually returns it to the tracer component. Finally, the tracer stores the extracted dispatch table into the data manager, so that other components can leverage this information for their operations.

Note that the size of the dispatch table is not known *a priori*, nor is contained in the parameters of the trapped function. To solve this problem, LynxFuzzer leverages the structured nature of such tables to infer how many entries it has and dump it. Indeed, each table entry is formed by: a function pointer, which must reside in the memory area of the target, and 4 consecutive integers, two of which must fall in the [0:16) range. Although not provably sound, such heuristics have indeed shown to be an effective and reliable technique to extract dispatch tables during our experiments.

Many heuristics can be used by the hypervisor to understand if an intercepted request is really the artificial one issued by the tracer. At first, the hypervisor checks if the dispatch table entry passed as a parameter lies in the memory range of the target kext, and if the selector value is 0. Furthermore, it is possible to specially craft the parameters of the artificial request to include an *a priori* specific pattern that the hypervisor can easily recognize. As for the previous heuristic, during our experiments LynxFuzzer correctly identified the artificial request for every target.

5.2.2 Sniffer

Many fuzzing techniques use a knowledge base, i.e., a set of valid input values, to build new inputs that are more likely to trigger bugs than randomly-generated ones. Simpler fuzzers, e.g., targeting pdf-viewers or protocols, commonly have a non-negligible number of valid inputs at their disposal to use as a starting point. Unfortunately, this is not our case, because we offer a generic framework to fuzz potentially unknown kexts, so we need to create our own base of knowledge before starting the fuzzing. For our strategies, the information gathered by the tracer are not enough, even if they already allow to considerably reduce the fuzzing space.

Indeed, beside the checks performed by IOKit, the kext itself could implement constraints on input values. Thus, `LynxFuzzer` includes a *sniffer* component, that is able to intercept real executions (i.e., not artificially triggered by our framework) of the target methods and to extract their parameters. To do so, we once again leverage `LynxFuzzer` hypervisor component that is able to *transparently* and *seamlessly* intercept whenever an interesting function is executed, and dump its parameters, by inspecting the target kext memory without altering it anyhow.

Intuitively, since the tracer has already retrieved the dispatch table, the sniffer could just ask to the hypervisor to place a breakpoint on every function contained in the table. However, such a solution would not be efficient. Thus, the hypervisor intercepts the `externalMethod()` function, whose parameters contain enough information to retrieve valid inputs, as can be seen in Figure 5.3. Indeed, the hypervisor uses the `dispatch` argument to discriminate which kext is the target of the intercepted invocation and `selector` to understand the method being called. If these parameters indicate that an intercepted invocation is not relevant (i.e., the callee is not the target of the fuzzing), then the hypervisor just ignores it. Otherwise, valid inputs are extracted from `IOExternalMethodArguments` structure containing the actual parameters that will be passed to the callee. Such data structure contains fields that allow to precisely infer the number and size of input parameters, so no heuristics are needed to read and send them back to the user-space sniffer component, which in turn stores them into the data manager.

The sniffer is almost essential when the check on parameters size is not performed by IOKit but delegated to the kext code (as happens when a dispatch table entry specifies a `0xffffffff` size for an input or output structure). In such a situation, the fuzzer does not even have minimum information on constraints commonly extracted by the tracer. Thus, having a set of valid inputs where to start from is of paramount importance, because otherwise, the fuzzing would be completely blind.

Please note that if `LynxFuzzer` is used to fuzz kexts for older, 32 bit, OS X version, then understanding which is the callee kext when intercepting an `externalMethod()` is not so straightforward, because the address of the dispatch table is not passed to this function. However, it is possible to slightly adapt our 64 bit strategy by moving the breakpoint *after* the body of the `externalMethod()` function retrieved the dispatch table and extract the address of the table. With this simple workaround, our sniffer component can deal also with older OS X versions.

To make the explanation clearer, previous paragraphs described the operation of *both* the tracer and the sniffer as if they were used to deal with a single kext. However, we remark that `LynxFuzzer` is actually capable of automatically identifying *every* kext loaded on an OS X system and then gather valid input values for all of them.

5.2.3 Fuzzer

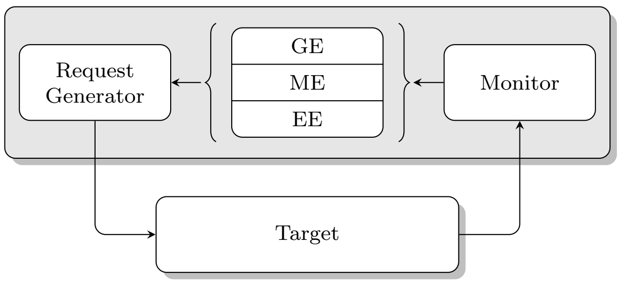


Figure 5.6: Fuzzer sub-components.

The fuzzer is the main component of LynxFuzzer. After the tracer and the sniffer obtained all the ancillary information needed to properly fuzz one (or more) kext, the fuzzer creates test cases (i.e., set of inputs) for the kext methods and invoke them through the IOKit device interface. Figure 5.6 reports the inner architecture of the component, that has *three* main parts: a request generator, a set of fuzzing engines and a monitor.

The *request generator* is an extremely versatile component: it must operate independently from the target kext *and* the selected fuzzing engine. In a typical execution, it receives a test case from the engine, checks that it respects what is specified in the dispatch table entry of the target method and properly crafts the argument for an `IoConnectCallMethod()` that eventually executes the target method in the kext.

If the test case does not cause a crash, the kext sends back an answer that is received by the *monitor*. Depending on the received answer and on the engine currently in use, the monitor may decide to alter the engine so that the next test case will depend on the result of the previous one. As we will see later in this section, both the *mutation engine* and the *evolution engine* leverage this information.

LynxFuzzer, furthermore, implements the concept of *session-based* fuzzing: we do not save just the request that *triggers* the bug, but we record *every* request that we make from the beginning of the fuzzing session. This practice is common when fuzzing stateful network protocols [10], but is also useful in our scenario. There are indeed kexts that maintain a “state” that is changed by a number of different fuzzing requests until an invalid state is reached and a bug is triggered. For this reason, recording *communication sessions*, instead of single requests, greatly improves the reproducibility of a bug, as also shown by our experiments in section 5.3. To record communication sessions between the fuzzer and a target kext, every request is stored in the data manager. However, this may be problematic as triggering a bug at kernel level usually corresponds to crashing the target system, on which the data manager is running, with possible losses of data. The simplest solution is to store the request *before* issuing it, to minimize the probability of losing it. Sometimes, however, it is possible that writes are not flushed to the disk before the crash. To avoid this, the data manager relies on databases, rather than files, that are much more resilient in dealing with crashes. This solution proved to be successful: in our experiments we did not observe any data loss. Should any be encountered, however, it is possible to implement a small cache in the hypervisor component, to store latest requests. Indeed, the hypervisor can survive even the worst crash of the guest and allow to safely dump cached requests, if needed.

Finally, the fuzzing engine is responsible for the production of input values (or *input vectors*) that will be used by the request generator. Details of the three different engines that we implemented in LynxFuzzer are described in the following sections.

Generation Engine (GE)

This is the simplest and quickest engine of the three. Its generation process may be summed up as follows. At first, it builds data structures that can contain the input for the target method. Then, it generates pseudo-random inputs and fill the structure that are then sent to the target, invoking the method through `IoConnectCallMethod()`. If the system does not crash, then the procedure begins anew. This engine does not need the monitor as the generation is random and does not take care of results of previous requests.

Please note that the generation engine is not *completely* random. Indeed, inputs are forged on top of the constraints extracted by the tracer. So, even if random, parameters that are generated by this engine are never discarded by IOKit internal checks.

The main advantages of this engine, with respect to the other we implemented, are its simplicity and low overhead.

Mutation Engine (ME)

This second fuzzing approach follows a principle that is the opposite of the previous one: every new input is generated from valid inputs collected by the sniffer component. The fuzzing process is roughly made of the following steps. Valid inputs that were previously gathered by the sniffer are *mutated* with different functions. Then, request containing such forged inputs are sent to the target method. If the system does not crash, the monitor checks the response of the kext, possibly excluding values that caused the kext to return an error from next mutations. This greatly increases the efficiency of the fuzzer, in the case of inputs structures with a variable size, because it gradually eliminates those that are not accepted because of checks performed *in* the code of the target method.

The set of mutation functions used by this engine includes: *bit flipping*, *byte flipping*, *byte swapping* and *size change*. The last one can be applied only to method whose dispatch table entry specifies a variable size.

The main advantages of this engine are its precision and its ability to gradually discard mutations that are rejected. Mutations, however, are random: the engine does not try to make any inference from the response of the method, except for discarding unaccepted values. Furthermore, the dependency on valid sniffed inputs may be too restrictive, depending on the usage scenario. In some cases, indeed, being able to collect a large base of valid inputs may take a long time.

Evolution Engine (EE)

The evolution engine tries to overcome the limitations of the previous ones. In an effort to reduce the use of pseudo-randomness, it leverages concepts of evolutionary algorithms to generate new inputs.

The hearth of any evolutionary algorithm is the *fitness* function, that depicts the fittest elements that will contribute to build new generations. In LynxFuzzer, we devised two different fitness functions: one that measures the code coverage of an input vector and one that measures the distance of an input from an ideal target vector. In the first case, we strive to create a set of input vectors that can give us the best code-coverage rate possible. The second, on the other hand, is useful when we want to individuate inputs similar to a given one (e.g., a vector that is known to trigger a bug in the target).

Independently by the fitness function selected, the evolution engine operates as follows. At first, it creates the initial population either from randomly-generated input vectors or from data gathered by the sniffer component, and defines the target vector (if any). For each vector in the current population it calculates the fitness value. Then, according to the fitness value and function used, the engine selects best candidates to contribute to the next generation (*selection* phase). There may be corner cases in which the target driver imposes strict internal checks on inputs, leading to a rapid degeneration of the population. To avoid this situation, we create an "elite group" of valid inputs with an high fitness from the initial population and propagate them through next populations. Couples of selected vectors are picked and used to generate a *new* vector (*crossing-over*). The "child" is generated by splitting the input vectors of both parents and randomly concatenating one part from each of the two. Random mutations (as those defined in the mutation engine) are applied to vectors generated by the previous step, to modify the genetic makeup inherited by the parents (*mutation* phase). Finally, the test cases in the new population are sent to the target. If needed, code coverage values for the new input vectors are calculated. The process starts again from the first step, by using the new population.

Code coverage based fitness. The first of the two fitness function that we implemented in LynxFuzzer is based on measuring the code coverage that is obtained by each single input vector. *Dynamically* measuring the code coverage of software running in kernel-space can be a cumbersome task, especially when the

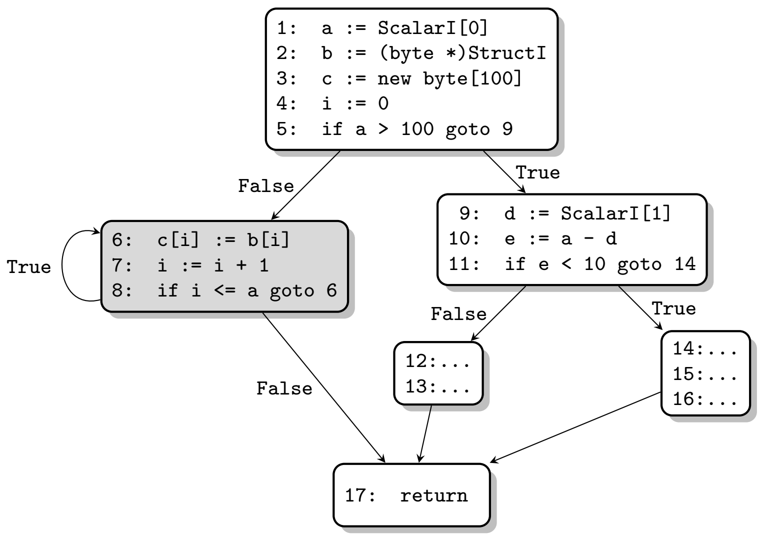


Figure 5.7: Example of weighted code coverage.

OS does not originally provide support for such analysis and, even if it did, one could argue that the mechanism itself could change the behavior of the target, thus invalidating the analysis. For this reason, such information are gathered by the hypervisor component, as explained in section 5.2.4.

Information collected by the hypervisor allows the fuzzer to operate on two dimensions: the set of distinct executed instructions and the number of times each instruction was executed. To understand why both are useful, consider the Control Flow Graph (CFG) [3] reported in Figure 5.7. Such CFG contains the pseudo-code of a simple method that receives two integers (`ScalarI[0]` and `ScalarI[1]`) and a structure (`StructI`) as parameters, and suffers from an off-by-one vulnerability in the slightly darker basic block. Considering only the number of executed instructions, this method would get a fitness value of 9, if `a` is less or equal to 100. Otherwise, the value will be 11 or 12, according to `d`. In such a scenario, input vectors that follow the first true branch would be more likely selected, sacrificing the branch that leads to the vulnerability. To prevent this situation, we decided to consider the second dimension too, and created a fitness function that assigns a higher weight to instructions that are executed multiple times.

Obviously, the “interesting” part of the code may also be under a branch that is highly disadvantaged by using weighted code coverage. For this reason, we decided to include both weighted and non-weighted fitness functions in LynxFuzzer, and use them alternatively.

Distance based fitness. The second approach is much more traditional. We assume to have an ideal input vector and the fitness function is based on calculating the *distance* between a generated vector and the target: the greater the distance, the lesser the fitness. To calculate the distance we consider each input as a vector of bytes $\{b_1, b_2, ..., b_n\}$, and we map it to a point $B = (b_1, b_2, ..., b_n)$ in an n-dimensional space. Given $V = (v_1, v_2, ..., v_n)$, the point corresponding to the ideal vector target, the fitness of B is calculated as the euclidean distance between the two points.

We identified three different input vectors that are useful when used as a target. The *Farthest Vector Target*, is a vector composed of all `0x00` or `0xff`, to make the fuzzer reach corner cases. The *Middle Vector Target*, composed by portions of vectors of the initial population, tries to maximize the number of generated input vectors that are not discarded by internal checks in the driver. Finally, the *Panic Vector Target* is a vector that is known to cause a panic in the target kext. This is particularly useful during debugging, where having multiple inputs that trigger the same bug can greatly help the fix process.

5.2.4 Hypervisor component

When fuzzing, it is very important to avoid inadvertent modifications of the target. Yet, as we saw, almost every component of LynxFuzzer needs to intercept certain IOKit functions or to gather some data from kernel memory. To be able to do this seamlessly, we decided once to leverage hardware-assisted virtualization technology [92]. In particular, we built the hypervisor component of LynxFuzzer on top of the dynamic analysis framework that we illustrated in chapter 3.

When loaded, the hypervisor first sets up EPT structures to create a one-to-one mapping between guest and host physical addresses, as we do not expect the guest system to try to attack us anyhow. Nonetheless, we write-protect internal hypervisor structures and code, since side-effects triggered by a bug may endanger the hypervisor integrity.

The main functionality that LynxFuzzer needs to implement at the hypervisor level is kext code coverage analysis. All other tasks, e.g., breakpoints to intercept kernel functions, are performed through the API of the framework.

Code coverage analysis. Performing code coverage analysis of a kernel component is a non trivial task. Intel offers a technology that can be leveraged to this purpose: Branch Trace Store (BTS) [56, 118]. BTS technology records, for each taken branch, interrupt, or exception, the originating address (i.e., current `eip`) and the target in a buffer inside the DS save area. When the buffer is full, an interrupt is triggered. Unfortunately, however, when BTS is enabled, the CPU enters a particular execution mode that automatically runs 25x to 30x slower than normal [55]. Furthermore, Soffa *et al.* analyzed that the slowdown can reach 40x, when BTS is used for branch coverage analysis [118]. This makes BTS ill-suited for LynxFuzzer, as such slowdown would destroy its effectiveness.

A faster, hardware-assisted, solution consists in leveraging Last Branch Record (LBR) [55]. LBR is similar to BTS in that it can be used to perform branch tracing. However, differently from the former technology, taken branches, interrupts, and exceptions are stored onto the LBR stack, whose size depends on CPU model, and ranges from 4 to 16 entries. Each entry contains the source and the destination address, thus these information can be used to calculate code coverage, just like with BTS. The LBR stack is filled circularly, and there is no interrupt mechanism to notify the analyzer that an overflow happened. This means that the analyzer must continuously poll the LBR stack to gather useful information. Even with very small sampling intervals, this approach cannot guarantee fully precise results, and a trade off between precision and performances must be found. Walcott *et al.* implemented THeME [139], a system that uses LBR to calculate code coverage. In their experiments, they evaluated precision and overhead of THeME with different sampling intervals. Precision is calculated as the number of instructions that are observed with LBR sampling over the number of instructions that are really executed. With the smallest interval (500 CPU cycles) THeME achieves an average 76% precision, with an extremely reasonable overhead (i.e., less than 100%). With a bigger sampling interval (i.e., 50M CPU cycles), the average precision lowers to 54% but THeME is faster (around 30% overhead). Such numbers show that LBR is a good solution when *approximate* code coverage information are acceptable but, unfortunately, this is not our case.

To overcome the problems of the aforementioned techniques, we devised a more efficient solution, based on EPT technology, that was not originally analyzed in Soffa *et al.*'s work [118]. Before starting to invoke kext methods, the fuzzer component communicates to the hypervisor (via hypercall) the range of code pages of the kext. The hypervisor removes the `EXEC` permission from the EPT entries corresponding to received memory ranges. As soon as the kext tries to execute code in such pages, it triggers an EPT violation and the hypervisor keeps track of the instruction that caused it. To proceed, it restores `EXEC` permission on the faulting page and configures the guest to perform a *single-step*. When the hypervisor gets the control back, because of the resulting debug exception, it removes the permission, so that the next instruction will violate again. Once the fuzzed methods returns, the fuzzer sends another hypercall to disable the tracing. The hypervisor stores collected information into a buffer in the fuzzer address space so that it can be used by the user-space component to calculate the code coverage corresponding to the invocation.

We evaluated the overhead of running the generation engine on each method of 10 different kexts, with and without the code coverage analysis enabled, and measured the slowdown. In particular, we first measured the number of requests per second that each module is able to serve while *not being analyzed*, then we collected the same measure while tracing, and compared the results. For the sake of precision, measurements were repeated 10 times for each module and results were averaged. The average overhead is 3.45x, with a best and worst case of, respectively, 1.73x and 5.99x. Detailed results of this evaluation are reported in Table 5.1. As we can see, we pay a high price for obtaining full precision without

Kernel Module	**Rps w/o Tracing**	**Rps w/ Tracing**	**Overhead**
`AppleSMCLMU`	668.28	385.90	1.73x
`AppleMikeyDriver`	517.54	283.28	1.83x
`AppleHDAController`	591.82	273.77	2.16x
`IOHDAFamily`	634.17	278.41	2.28x
`AppleSMC`	651.99	284.31	2.29x
`IOSurface`	434.84	112.44	3.87x
`AppleHDA`	635.39	164.20	3.87x
`IOUSBFamily`	204.89	42.76	4.79x
`SimpleDriver`	424.91	74.01	5.74x
`IOHIDFamily`	278.00	46.44	5.99x
Overall	—	—	3.45x

Table 5.1: Overhead of the code coverage analysis.

modifying the target, if compared to instrumentation [118] or THeME [139]. However, consider that the only approach that can potentially obtain the same results of our technique is BTS. With an overhead of 40x [118], BTS is more than one order of magnitude slower than our approach. Furthermore, consider that BTS and LBR usually slow down the *entire system* on which the target runs, while our approach is as precise as instrumentation, and only slows down the target of the analysis (and whomever is interacting with the it at the same time).

5.3 Experimental evaluation

This section presents the result of the experiment we conducted to evaluate the effectiveness of LynxFuzzer. To this purpose, we used the fuzzer to exercise a set of 17 different Kernel Extensions and found bugs in 6 of them[2]. More detailedly, 2 of the 6 have already been patched in OS X 10.9 (Maverick) and have been assigned the following CVE-ID by Apple: CVE-2013-5166 and CVE-2013-5192 [5]. The remaining 4 are still unpatched and will be most likely addressed in the next releases of OS X.

All the experiments were conducted on a Mac OS X 10.8.2 system (Mountain Lion), installed on Apple hardware, equipped with an Intel i5 CPU and 12GB of RAM. Thanks to the adoption of kernel-security measures in OS X, none of the bugs we identified can be easily exploited to perform a privilege escalation attack.

Unfortunately, we cannot yet disclose details of the bugs, if not that Apple acknowledged all 6 of them and already fixed 2 in the to-be-released OS X 10.9

[2]Obviously, all the bugs were reported to Apple or the appropriate vendor.

Kext	**Code Coverage Perc.**		
	Methods	**Estimation**	**Full**
`IOUSBFamily`	64.8%	61.1%	33.7%
`IOHIDFamily`	86.4%	69.9%	11.4%
`SimpleDriver`	96.6%	77.3%	34.3%
`IOSurface`	76.6%	58.8%	18.5%
`AppleUSBHub`	86.9%	54.5%	37.5%

Table 5.2: Coverage analysis results.

(Maverick). The remaining 4 will be addressed in next releases, and we were requested to keep all of them confidential in the meantime.

A metric that is usually associated with efficiency of a fuzzer, is the *code coverage* level. However, as also stated in [49], such metric may not be extremely significant: a fuzzer could even reach 100% code coverage of the analyzed code, but yet fail at finding a bug. Consider once again the example reported in Figure 5.7, where two input integers determine the flow of execution. If used as an input source, the following set of values $I = \{(10, 3), (120, 100), (120, 115)\}$, respectively assigned to `ScalarI[0]` and `ScalarI[1]`, would guarantee a code-coverage of 100%, without ever triggering the off-by-one vulnerability. Nonetheless, since it is customary to report such information, we conducted an experiment to calculate the code coverage level of LynxFuzzer.

Even if our hypervisor component can easily keep track of every instruction that is executed in a given time-frame (e.g., during the fuzzing), giving a *precise* measure of the coverage of a kext is quite challenging. The problem lies in understanding exactly *how many* instructions of a kext can be reached by user-space programs when invoking methods exported in the dispatch table, to use them as a baseline to estimate a percentage of the code coverage. The simplest solution would be to statically count the instructions that constitute such methods. This measure, however, would be imprecise, because it does not keep track of all the ancillary functions that are invoked by exported methods. On the other hand, considering the whole code of the kext as the maximum amount of coverable code is imprecise as well, because it would mean to include all the methods that are invoked at load-/unload-time or during the the *Driver Matching* phase, (i.e., all the *life-cycle* methods [70]), thus leading to an underestimation of the obtained coverage.

To overcome these limitations and give an estimation of the amount of code that can be reached from methods exported in the dispatch table we use, once again, an hybrid static-dynamic technique. First, we statically count the instructions of the exported methods and individuate all the control transfer instructions (CTI). Then, for each CTI, if the target is another method of the same kext, we add its instructions to the overall count.

Unfortunately, this is not enough because, due to their object-oriented nature, kexts include a high number of *indirect* CTIs, that cannot be followed statically. For such instructions, we revert to *dynamic* analysis: we modified LynxFuzzer hypervisor so that it dumps the target of every indirect CTI executed in the kext code. If the target corresponds to a method of the kext that had not been deemed reachable by the static analysis, then we update the instructions count with the newly discovered method. Obviously, dynamically analyzing *indirect* CTIs only allows us to see a subset of all the possible targets, thus it is impossible to gain a fully precise count, but yet it allows us to refine our previous estimation.

Table 5.2 reports the code coverage results of a subset of the fuzzed kexts. Under the "Coverage" column we report three different code coverage percentages, respectively calculated over: the number of instructions of exported methods, the static/dynamic estimation we just described and, finally, the overall number of instructions contained in the kext. As we already said, consider that the first value is an *overestimation* of the coverage, while the last is an *underestimation*.

The values we reported have been obtained by measuring the code coverage level of the three different fuzzing engines considered as a whole. We decided to report only this aggregated value because our purpose is to show the efficiency of our fuzzing framework, not of the fuzzing techniques we implemented inside LynxFuzzer.

5.4 Conclusion

LynxFuzzer is a fuzzing framework for Mac OS X kernel extensions. It is built on top of the hardware-assisted hypervisor framework that we presented in chapter 3 and uses three different input-generation engines to discover bugs that could lead unprivileged users to crash the machine or to attempt privilege-escalation attacks. We used LynxFuzzer to fuzz 17 kernel extensions on Mac OS X Mountain Lion, and discovered bugs in 6 of them.

The main problem we face with LynxFuzzer is that we are missing a precise measure of its efficiency, if not for the bugs we discovered. At the moment, indeed, our esteem of the code coverage obtained by our fuzzing activities is a best effort. Even if we are able to precisely track every instruction executed by the invocation of a kext method, the overall count over which we calculate the coverage percentage is approximate. Furthermore, the count is extracted statically (even for dynamically identified methods) and may be hindered by obfuscation in kext code. We did not experience this issue in the 17 kexts we analyzed. Nonetheless, we plan to find a better solution to improve the evaluation of our fuzzer.

As a future work, we plan to extend LynxFuzzer to enable testing of iOS kexts. This mobile OS, indeed, uses the same user-to-kernel interface of its desktop counterpart. Furthermore, despite the raise of Android, iOS still occupies a significant slice of the mobile market and, given the complete lack of “biodiversity” between devices running this mobile OS, every kernel bug could have a very strong impact.

6 Related literature

6.1 Dynamic analysis

During the years, a plethora of different methods and techniques to perform dynamic analysis at different levels (user, kernel, hypervisor) have been proposed by both research and industry. In this section we discuss those that we feel to be strictly related to the hypervisor-based framework we introduced in chapter 3.

6.1.1 Dynamic kernel instrumentation

Instrumentation typically entails complementing a piece of code with functionalities that were not originally provided (e.g., a debug print where no debug code is present). It is often used to monitor and analyze performances, to identify errors and to track the execution of different components, with different granularity levels: from a single user-space program to the whole kernel of an operating system. There exist many typologies of instrumentation that can be applied to source or compiled code, both *statically*, i.e., modifying the program before it is executed, and *dynamically*, i.e., inserting instrumentation code at run-time.

DTrace is a facility included into Solaris kernel that allows dynamic instrumentation of production systems [21]. The key points of DTrace are efficiency and flexibility. First, the instrumentation framework itself introduces no overhead. Second, the framework provides tens of thousands of instrumentation points. Actions to be taken are expressed through a high-level control language, similar to C, that enables a great number of actions and includes mechanisms to guarantee run-time safety. In particular, DTrace performs different checks on specified actions and intercepting errors that happen at run-time (e.g., illegal memory accesses). By doing so, it prevents actions from introducing fatal errors into the analyzed systems. Leveraging instrumentation points offered by DTrace, it would be possible to offer the same analysis functionalities of our framework. However, DTrace suffers from a strong limitation, due to the requirement that instrumentation points exist *a priori* in the kernel. Clearly, this makes the instrumentation framework extremely dependent on the target kernel and impossible to use on systems that do not offer such hooks. Our framework, on the contrary, does not suffer from these limitations, since it allows to analyze *any* software component, without relying on its source code neither on preexisting hooks.

KernInst is a dynamic instrumentation framework for commodity kernels [131]. To use KernInst it is not necessary to modify, or recompile, the target kernel. To offer instrumentation capabilities, the framework make use of an user-space demon and a kernel module, that must be installed on the target system. Users of KernInst can use a C++ API that simply communicates with the demon and hides details of instrumentation. Many different applications were implemented using KernInst API. Most notably, Kperfmon is a tool to gather precise data over kernel performances. The use of a user-space demon and of a kernel module allows KernInst to be used *without* modifying the kernel *a priori*, contrarily to DTrace. However, this does not make it independent from the analyzed system or easily usable with closed source systems. Indeed, the module that offers instrumentation functionalities must be implemented for each target kernel. Furthermore, its developer needs to have access to the kernel source code, or at least to a great deal of internals, because the module must insert instrumentation points in the right places at run-time, and such places are not always easy to find without a deep knowledge of the target. Please note that, as described in chapter 3, even our framework needs a small kernel module to be loaded on the target system. However, the implementation of the loading driver of our framework is trivial, as it just needs to load the code of the hypervisor, while the analysis code itself is contained in the hypervisor and almost completely OS independent. Another disadvantage of KernInst with respect to our framework is that its components are easily detectable.

6.1.2 Kernel-level debugging

Several efforts have been made to develop efficient and reliable kernel-level debuggers. Indeed, these applications are essential for many activities, such as the development of device drivers. One of the first and most widely used kernel-level debuggers that targeted the Microsoft Windows operating system was SoftICE [119], but today the project has been discontinued. However, both commercial [128] and open-source [108] alternatives to SoftICE appeared. Modern versions of Windows already include a kernel debugging subsystem [85]. Unfortunately, to exploit the full capabilities of Microsoft's debugging infrastructure, the host being debugged must be physically linked (*e.g.,* by means of a serial cable) with another machine. All these approaches share a common factor: to debug kernel-level code, they leverage another kernel-level module. Obviously, this is like a dog chasing its tail. The framework proposed in chapter 3 of this dissertation does not require any kernel support nor to modify the kernel to add the missing support at run-time.

6.1.3 Frameworks based on virtual machines

Instead of relying on a kernel-level module to monitor other kernel code, an alternative approach consists in running the target code inside a virtual machine and performing the required analyses from the outside [43, 12, 106, 144, 78].

In [62, 143, 34] the authors propose virtual machines with execution replaying capabilities: a user can move forward and backwards through the execution history of the whole system, both for debugging and for understanding how an intrusion took place. More particularly, King *et al.* [61] propose an execution replaying-based debugger that allows users, among other things, to set breakpoints on already executed instructions and to perform backwards single-stepping in the execution history of the system. Instead, ReVirt by Dunlap *et al.* [34], is a framework that tries to solve the problem of gathering consistent and reliable logging traces on systems compromised by malware with enough execution privileges to corrupt the kernel of the victim to hide their traces. ReVirt executes the target system inside a virtual machine and logging from outside, to prevent a malware from destroying its traces. Furthermore, it leverages execution replaying techniques to grant an extreme precision and completeness of the produced traces. Finally, Chow *et al.* propose Aftersight [25], a system that decouples execution recording from execution trace analysis, thus reducing the overhead suffered by the system where the guest operating system is run. Nowadays, Aftersight is part of the VMware platform, and other mainstream commercial products provide similar capabilities. The framework proposed in chapter 3 can provide these functionalities even on systems not running in a virtual machine natively. Aftersight is part of VMware [136].

All the approaches reported in this section are affected by two main problems. First, they assume the transparency of software hypervisors they use, and this is not always the case [81, 80]. Second, they require that the target system is natively bootstrapped as a guest of their virtual machines. This clearly prevents to use such approaches on systems that require particular hardware configurations or that cannot easily be rebooted inside a virtual machine.

6.1.4 Aspect-oriented programming

Aspect-oriented programming is a paradigm that promises to increase modularity by encapsulating cross-cutting concerns into separated code units, called “aspects”, whose “advice” code is woven into the system automatically, by specifying the properties of the join-points. AspectC is an aspect-oriented framework that is used to customize (at compile-time) operating system kernels [29, 74, 75]. More dynamic approaches have been proposed: for example TOSKANA provides *before*, *after* and *around* advices for in-kernel functions and supports the implementation of aspects themselves as dynamically exchangeable kernel modules [36]. The framework proposed in chapter 3 allows to achieve the same goal while being transparent and fault-tolerant.

6.1.5 Malware analysis

Dinaburg *et al.* proposed Ether [33], a malware analysis framework that has some common points with the framework we presented in chapter 3. Indeed, Ether leverages hardware-assisted virtualization to perform its analyses for an environment that is more privileged than the operating system where the malware runs and thus is transparent. These are fundamental features for a malware analysis framework, as we outlined many times in this dissertation. The main differences between Ether and our framework is that the former does not include an ad-hoc hypervisor, but is based on Xen [11]. Xen does not support late launching, restricting Ether analyses to systems that are natively run in a Xen virtual machine.

6.2 Malware detection

Most relevant papers for the AccessMiner detector and its uses of system-centric access activity models, which are illustrated in chapter 4, focus on malware detection at the system call and the system library interfaces. These interfaces best describe the system resources manipulated by a program (e.g., files, other programs, other processes, configuration data, authentication and authorization information, network communication channels), making system call-based detectors comparable to AccessMiner's access activity model.

Malware detection has looked at many ways to describe program behavior, and corresponding models evolved to keep pace with the increasing complexity of malware. Early detection mechanisms were based on particular byte sequences in the program binary that were indicative of malware. Over time, obfuscation strategies pursued by malware writers forced detectors to move to regular expressions over bytes [130], and eventually rendered them obsolete as byte patterns have little predictive power (i.e., they can accurately capture only previously seen malware). Other models such as byte n-grams [71], system dependencies of the program binary [112], and syntactic sequences of library calls [142, 90] have been proposed with limited success. AccessMiner emphasis on a system-centric approach to modeling resource interactions bypasses such syntactic artifacts.

The software-diversity tactics employed by malware writers required new detection techniques that could capture more of the intent of the program and less of the syntactic characteristics of the program binary. The research efforts have focused on describing malware in terms of violations to an information-flow policy. Because it is not feasible for performance reasons to track system-wide information flows accurately, the focus shifted on better and better approximations of the information flow. Bruschi *et al.* [16] and Kruegel *et al.* [66] showed that some classes of obfuscations could be rendered innocuous by modeling programs according to their instruction-level control flow, while Christodorescu *et al.* [26] and Kinder *et al.* [59] built obfuscation-resilient detectors based on instruction-level information flow. Nonetheless, instruction sequences are fungible and there are many ways to implement the same high-level functionality. Detection techniques then raised the bar by capturing information flow at the level of library calls, as proposed by Kirda *et al.* [63], system calls, as proposed by Kolbitsch *et al.* [64], Christodorescu *et al.* [27], Martignoni *et al.* [82], and Stinson *et al.* [122], and OS resources, as proposed by Yin *et al.* [145]. The respective evaluations of each of these techniques shows that as the models used in detection more closely describe actual OS resources, the detection rates significantly increase and the false-positive rates decrease. Unfortunately the library and system-call interfaces are rich enough that mimicry attacks are still possible [65, 138]. This observation guided the choice of using system resources as the basic element in our models, discarding any information about the order in which resources are accessed. Furthermore we focus strictly on system resources that are shared across processes (i.e., files, registry, network connections) and we ignore single-process resources such as virtual memory.

Beyond proposing a richer, system-centric model of program behavior, we made a concerted effort to improve an often overlook evaluation aspect, the external validity of the experimental settings. This concerns the number and diversity of benign and malicious programs used to evaluate a detection technique, as well as the environment in which they are exercised (in the case of detectors that rely on runtime information). For example, Kirda *et al.* [63] evaluated their system against 33 malware samples and 18 benign samples, each samples executed for 30–60 seconds. Kolbitsch *et al.* [64] used 563 malware samples and 10 benign samples, executed for up to 5 minutes. Christodorescu *et al.* [27] evaluated 16 malware samples and 6 benign samples for up to 4 minutes, similar to the test sets used by Martignoni *et al.* [82] (7 malware, 6 benign) and to Stinson *et al.* [122] (6 malware, 9 benign). Yin *et al.*, in their Panorama system, evaluated 42 malicious samples and 56 benign ones, for 5 minutes. What is common to all of these evaluations is that both the numbers of malicious samples and of benign samples are quite small. On current systems, regular users often run tens of interactive applications and hundreds of background processes, casting doubt on the relevance of results obtained from a few benign samples. Furthermore, evaluations in previous work were performed in virtualized, constrained environments, where interactive applications were exercised mechanically in ways that do not necessarily reflect real-life usage. We addressed these limitations by collecting execution traces of benign applications from actual users, during the course of their normal interaction with their personal systems. We designed our system to have low overhead and to anonymize all collected information, so that the users had no concerns and were not impacted in their regular use of their machine.

6.3 Fuzzing and vulnerability identification

The fuzzing approach we illustrated in chapter 5 has a rich related literature. Researchers over the years proposed many works focused on kernel protection, on the identification of bugs in kernel-level code or, more generically, on fuzzing and automatic testing techniques.

The closest work to LynxFuzzer is found in an online presentation by Xiaobo and Hao [23] that briefly analyzes the possibility of fuzzing OS X kernel extensions through *DeviceInterface*. Unfortunately, a deep comparison between such work and LynxFuzzer is hindered by the lack of details of the former. The depicted approach appears to be largely manual and there is neither an analysis of performances nor results of conducted fuzzing activities to compare.

Another similar, yet extremely more specific, approach is presented by Keil and Kolbitsch [58] who propose a stateful fuzzer for 802.11 device drivers. Beside its usage of stateful fuzzing, LynxFuzzer is quite different from this work, since it addresses generic kernel extensions and leverages hardware-assisted virtualization instead of emulation.

SLAM is a model checking engine whose goal is to automatically verify if a program correctly uses external libraries [9, 28]. On top of such engine, Microsoft created Static Driver Verifier, a tool to automatically analyze the source code of Windows device drivers and determine whether it correctly interacts with the Windows kernel. SLAM has many limitations, if compared to LynxFuzzer. First, it requires the source code of the drivers it analyzes. Second, its scope is limited to the correctness of interactions with known libraries. Finally, users of SLAM must feed it with manually created rules that the tool will check for the verification, thus requiring some manual effort.

A technique often used for testing and verification is *symbolic execution* [60], along with its hybrid static-dynamic spin-off, i.e., *concolic execution* [114], that we briefly illustrated in chapter 2. There are a plethora of works based on both these techniques, but latest efforts focused more on the latter, as it allows to—at least partly—bypass the *path explosion* problem. Among these, S^{2}E is of particular interest for this dissertation [24]. S^{2}E is a binary analysis framework that implements the idea of *selective symbolic execution*, a method to minimize the amount of code that needs to be executed symbolically. Users of S^{2}E are able to select particular portions of the target, and these portions are the only that will be executed symbolically, while all the remaining code will be executed concretely. Selected portions are translated into LLVM bytecode [69] and executed symbolically through a modified version of the KLEE engine [18]. Furthermore, on top of S^{2}E, authors built DDT$^+$ to automatically discover bugs in Windows device drivers. Even if remarkably useful, S^{2}E has some drawbacks. Selecting only small portions of code to test is extremely useful when performed by a developer who is testing its own code and, thus, has a more or less precise idea of where to look for bugs. In less comfortable scenarios, as for LynxFuzzer, we could reduce the code to be executed symbolically to a single kext at a time, or maybe even discard some of the kext methods, but selected portions would still be considerably large. With a runtime overhead of 78x for symbolic execution, analyzing large portions of code could be unbearably long. This problem has also been reported by Dowser [49] that uses S^2E for its symbolic execution and tested it with a small—yet fairly complex—piece of code containing a buffer underrun. The vulnerability was discovered by S^2E only after authors manually reduced the number of symbolic variables to one small string. Without such reduction, it was unable to find the vulnerability in 8 hours, thus, even if reduced, the path explosion problem is still consistent.

Dowser is probably the most advanced approach to fuzzing targeted at finding buffer overflows [49]. It uses a mix of static program analysis, concolic execution, and taint tracking to automatically steer the execution of a program to interesting locations, more likely to contain a buffer overflow. In particular, the observation behind Dowser is that there are particular sets of instructions that are more error-prone than others (e.g., pointer arithmetic). Interesting instructions are identified through static analysis with LLVM and, later, the program is driven to these instructions through concolic execution. To avoid path explosion, Dowser proceeds as follows: first it aims at maximizing pointer value coverage, rather than pure code coverage; second, it uses taint analysis to identify which portions of the input can influence memory accesses in targeted code areas, and treats only such portions as symbolic. Dowser is extremely powerful, and LynxFuzzer could benefit by adopting a similar approach to its fuzzing activities. Unfortunately, our approach is aimed at testing closed source software, while Dowser relies on the source code of the target to locate interesting instructions.

Finally EXE [19] and SmartFuzzer [68] are similar to LynxFuzzer, as they both use a combination of static and dynamic analysis to automatically generate inputs vector that can efficiently exercise user-space applications.

7 Conclusions and future directions

In this dissertation we have investigated how hardware-assisted virtualization constitutes an optimum basis to tackle three different problems in systems security.

As a first contribution, we presented a generic framework to perform complex dynamic analyses of both user- and kernel-level software. Leveraging hardware-assisted virtualization, our framework is transparent to the target of the analysis, it does not modify the target, and does not rely on any native support for the analysis. These properties allow to overcome many issues of previous dynamic analysis approaches. Furthermore, our framework offers an API that can be leveraged to build custom analysis tools that will be executed on top of the framework. As a further contribution, we implemented HyperDbg, a kernel debugger that can be used to dynamically analyze the properties of three different operating systems: Windows (XP and 7), Linux and Mac OS X.

The second contribution is a *malware detector*, built as an evolution of AccessMiner [67]. AccessMiner uses a novel *system-centric* technique to model the activity of benign programs. With this technique it is possible to identify common patterns of how benign programs interact with the operating system resources. Such patterns are used to create *access activity models* that can be later leveraged to identify malicious programs whose accesses are not in the model. With a reasonable amount of training, such models have a very high detection rate with *no false positives*. The contribution of this dissertation is the description of a *new version* of AccessMiner, and how it can be implemented on top of our hypervisor-based framework. This proved to be quite challenging, as we needed not only to create a resilient and transparent system, able to perform the fine-grained analysis required to intercept and inspect resource accesses, but, at the same time, we needed to keep performances in great consideration. Indeed, the new AccessMiner detector module is built to be an active malware detector and, thus, must not have a big impact on performances of the protected system.

The third, and last, contribution is the design and implementation of Lynx-

Fuzzer, a tool to automatically find vulnerabilities in Mac OS X kernel level components. LynxFuzzer, as suggested by its name, is based on fuzzing and implements three different input generation engines to stress test kernel extensions. LynxFuzzer leverages hardware assisted virtualization to inspect tested kernel extensions, without modifying them, a strong prerequisite of fuzzing: any inadvertent modification of the target, indeed, may easily lead to wrong results. LynxFuzzer has been implemented and tested on Mac OS X 10.8.2 (Mountain Lion), and provided unexpected results, thus proving its usefulness and effectiveness. More precisely, we individuated bugs in 6 of the 17 kexts we analyzed. Furthermore, both to support one of the fuzzing engine, and to give a partial evaluation of our fuzzer effectiveness, we implemented a code coverage analysis techniques that gave interesting results, in terms of obtained precision and overhead compared to existing techniques that can be used to measure the code coverage of kernel-level components.

New horizons opened by hardware-assisted virtualization are not limited to what has been shown in this dissertation. At the moment, indeed, we are already investigating how to leverage EPT to fine-grainedly track memory allocations of a process, without dwelling on the many details and intricacies of specific allocators. The ability to track how a process deal with memory, abstracting from the allocator used by the system, is interesting because it may allow to detect, and block, some exploits. In particular, modern exploits targeting browsers and pdf readers use spraying techniques [14] to bypass ASLR [133], by filling memory with executable attacker-controlled code to maximize the probability of hijacking the control flow to a useful memory location, despite randomization. Preliminary results show that tracking a process allocations and its write-then-execute patterns on allocated pages can identify attacks with a good precision, and a fairly low overhead. Results are promising, but we still need to thoroughly test our approach, and tune our detection heuristics to find a wider range of attack variations.

Finally, nowadays it is impossible to speak of systems security without at least hinting at *mobile* systems security. Such systems are indeed becoming more and more frequently a target for cybercriminals and their characteristics make it hard to adapt previously developed solutions. For example, reduced computing power and low battery life require to inspect solutions with an extremely low impact on these resources, making static analysis attractive once again. Software virtualization, however, has been already to apply dynamic analysis on mobile applications [106, 144]. Such solutions, however, are still confined to offline analysis, because of the amount resources they require. Recently, however, ARM introduced hardware support for virtualization on its CPUs too (most of the CPUs used on mobile devices are ARM) [46]. Thus, we plan to investigate if, and how, ARM hardware-assisted virtualization technology can be used to develop powerful defense and prevention techniques for mobile operating systems, keeping in mind the scarce resources of mobile devices.

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *USENIX Summer*, 1986.

[2] Keith Adams. Blue Pill detection in two easy steps, 2007.

[3] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*, volume 1009. Pearson/Addison Wesley, 2007.

[4] AMD, Inc. AMD virtualization. `http://www.amd.com/virtualization`.

[5] Apple. About the security content of OS X Mavericks v10.9, October 2013. `http://support.apple.com/kb/HT6011`.

[6] Apple Inc. I/O Kit Fundamentals. `https://developer.apple.com`.

[7] Daniel Ashlock. *Evolutionary computation for modeling and optimization*. Springer Science+ Business Media, 2006.

[8] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, 2011.

[9] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *Integrated formal methods*, pages 1–20. Springer, 2004.

[10] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: toward a stateful network protocol fuzzer. In *Information Security*, pages 343–358. Springer, 2006.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM symposium on Operating Systems Principles*, New York, NY, USA, 2003.

[12] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A tool for analyzing malware. In *Proceedings of the Annual Conference of the European Institute for Computer Antivirus Research*, Hamburg, Germany, 2006.

[13] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of USENIX Annual Technical Conference*, Anaheim, California, USA, 2005.

[14] Dion Blazakis. Interpreter exploitation: Pointer inference and jit spraying. *BlackHat DC*, 2010.

[15] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified Process Replicae for Defeating Memory Error Exploits. In *Proceedings of the 3rd International Workshop on Information Assurance*, New Orleans, Louisiana, USA, 2007.

[16] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *Proceedings of the 3rd Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, Berlin, Germany, 2006.

[17] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *Proceedings of 20th USENIX Security Symposium*, San Francisco, California, USA, 2011.

[18] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the* 8th *Symposium on Operating Systems Design and Implementation*, San Diego, California, USA, 2008.

[19] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[20] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, Minneapolis, MN, USA, 2012.

[21] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX Annual Technical Conference*, Boston, Massachusetts, USA, 2004.

[22] Xu Chen, Jonathon Andersen, Zhuoqing Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the 38th Conference on Dependable Systems and Networks*, Anchorage, Alaska, USA, 2008.

[23] Xu Hao Chen Xiaobo. Find Your Own iOS Kernel Bug, 2012.

[24] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, California, USA, 2011.

[25] Jim Chow, Tal Garfinkel, and Peter Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of USENIX Annual Technical Conference*, Boston, Massachusetts, USA, 2008.

[26] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2005.

[27] Mihai Christodorescu, Christopher Kruegel, and Somesh Jha. Mining specifications of malicious behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, 2007.

[28] Edmund M Clarke, Orna Grumberg, and Doron A Peled. *Model checking.* MIT press, 1999.

[29] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European Software Engineering Conference*, Vienna, Austria, 2001.

[30] Intel Corporation. Performance evaluation of intel ept hardware assist, 2009. `http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf`.

[31] Saumya K Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems*, 22(2):378–415, 2000.

[32] Fabrice Desclaux and Kostya Kortchinsky. Vanilla Skype, 2006.

[33] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, Alexandria, Virginia, USA, 2008.

[34] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementations*, Boston, Massachusetts, USA, 2002.

[35] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2):6, 2012.

[36] Michael Engel and Bernd Freisleben. TOSKANA: A toolkit for operating system kernel aspects. *Transactions on Aspect-Oriented Software Development II*, 4242:182–226, 2006.

[37] Natalie Evans. Barclays: Eight men arrested over 1.3million theft in latest bank cyber crime plot, 2013. `http://www.mirror.co.uk/news/uk-news/barclays-eight-men-arrested-over-2287318`.

[38] Aristide Fattori, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. Hypervisor-based Malware Protection with Accessminer. *Submitted to the Journal of Computer Security*, 2013.

[39] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, Antwerp, Belgium, September 2010.

[40] Peter Ferrie, Nate Lawson, and Thomas Ptacek. Don't tell Joanna, the virtualized rootkit is dead. Black Hat USA, 2007.

[41] CrySyS Lab Gábor Pék. Evading Intel VT-d protection by NMI interrupts Security Advisory, 2013. `http://goo.gl/2hMsIh`.

[42] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, San Diego, California, USA, 2007.

[43] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Symposium on Network and Distributed Systems Security*, San Diego, California, USA, 2003.

[44] Patrice Godefroid, Michael Y Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the 16th Network and Distributed System Security Symposium*, San Diego, California, USA, 2008.

[45] Robert P Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[46] John Goodacre. Hardware accelerated virtualization in the ARM cortex processors. In *XenSummit Asia*, November 2010.

[47] James Green. Intel trusted execution technology. Technical report, Intel Corporation, 2012.

[48] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, et al. Manufacturing compromise: the emergence of exploit-as-a-service. In *Proceedings of the 19th Conference on Computer and Communications Security*, 2012.

[49] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of 22nd USENIX Security Symposium*, Washington, DC, USA, 2013.

[50] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: a scalable system for detecting code reuse among android applications. In *Proceedings of the 10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, Berlin, Germany, 2013.

[51] Sean Heelan. *Automatic generation of control flow hijacking exploits for software vulnerabilities.* PhD thesis, University of Oxford, 2009.

[52] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows kernel.* Addison-Wesley Professional, 2005.

[53] T. Holz, M. Engelberth, and F. Freiling. Learning more about the Underground Economy: A Case-Study of Keyloggers and Dropzones. In *Proceedings of the 14th European Symposium on Research in Computer Security*, Saint Malo, France, 2009.

[54] Intel Corporation. Intel Virtualization Technology for Directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices, 2012. `http://goo.gl/C0IYnC`.

[55] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A*, March 2013.

[56] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, March 2013.

[57] Iseclab. Anubis. `http://anubis.iseclab.org`.

[58] Sylvester Keil and Clemens Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan*, 2007.

[59] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *Proceedings of the 2nd Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, Vienna, Austria, 2005.

[60] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[61] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2006.

[62] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of USENIX Annual Technical Conference*, Anaheim, California, USA, 2005.

[63] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[64] C. Kolbitsch, P. Milani, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium*, Montréal, Canada, 2009.

[65] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, Maryland, USA, 2005.

[66] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, Seattle, WA, 2005.

[67] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. AccessMiner: Using system-centric models for malware protection. In *Proceedings of the 17th Conference on Computer and Communications Security*, Chicago, Illinois, USA, 2010.

[68] Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A smart fuzzer for x86 executables. In *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*, Minneapolis, Minnesota, USA, 2007.

[69] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, San Jose, California, USA, 2004.

[70] Jonathan Levin. *Mac OS X and IOS Internals: To the Apple's Core*. Wrox, 2012.

[71] Wei-Jen Li, Ke Wang, Salvatore J. Stolfo, and Benjamin Herzog. Fileprints: Identifying file types by n-gram analysis. In *Proceedings of the 6th Annual IEEE Systems, Man, and Cybernetics Workshop on Information Assurance*, The Big Island, Hawaii, USA, 2005.

[72] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, Washington, District of Columbia, USA, 2003.

[73] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[74] Daniel Mahrenholz, Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. An aspect-oriented implementation of interrupt synchronization in the PURE operating system family. In *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems*, Málaga, Spain, 2002.

[75] Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Program instrumentation for debugging and monitoring with AspectC++. In

Proceedings of the Symposium on Object-Oriented Real-Time Distributed Computing, Washington, District of Columbia, USA, 2002.

[76] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, Minnesota, USA, 2007.

[77] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference*, Miami Beach, Florida, USA, 2007.

[78] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Ottawa, Canada, September 2010.

[79] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *Proceedings of the 7th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, Bonn, Germany, 2010.

[80] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU emulators. In *Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA)*, pages 261–272, Chicago, Illinois, USA, July 2009. ACM.

[81] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing system virtual machines. In *Proceedings of the 2010 International Symposium on Testing and Analysis (ISSTA)*, Trento, Italy, 2010.

[82] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C. Mitchell. A layered architecture for detecting malicious behaviors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, Boston, Massachusetts, USA, 2008.

[83] McAfee. McAfee threats report: Second quarter 2013, 2013. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2013-summary.pdf`.

[84] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems*, Glasgow, Scotland, 2008.

[85] Microsoft Corporation. Debugging tools for Windows.

[86] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 1990.

[87] Barton Paul Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services.* University of Wisconsin-Madison, Computer Sciences Department, 1995.

[88] Heidi Moore and Dan Roberts. AP Twitter hack causes panic on Wall Street and sends Dow plunging, 2013. `http://www.theguardian.com/business/2013/apr/23/ap-tweet-hack-wall-street-freefall`.

[89] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, Las Vegas, Nevada, USA, 2001.

[90] Srinivas Mukkamala, Andrew Sung, Dennis Xu, and Patrick Chavez. Static analyzer for vicious executables (SAVE). In *Proceedings of the 20th Annual Computer Security Applications Conference*, Tucson, Arizona, USA, 2004.

[91] Carey Nachenberg. Understanding and managing polymorphic viruses. Technical report, Symantec, Inc., September 1996.

[92] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.

[93] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[94] James Newsome, David Brumley, and Dawn Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, 2006.

[95] NIST. CVE and CCE statistics. `http://web.nvd.nist.gov/view/vuln/statistics`.

[96] NIST. CVE-2010-1794, 2010. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-1794`.

[97] NIST. CVE-2013-0109, 2013. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0109`.

[98] NIST. CVE-2013-0976, 2013. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0976.

[99] NIST. CVE-2013-0981, 2013. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0981.

[100] Northsecuritylabs. HyperSigh rootkit detector. http://northsecuritylabs.com/downloads/whitepaper-html/.

[101] Roberto Paleari. *Dealing with next-generation malware.* PhD thesis, Università degli Studi di Milano, 2010.

[102] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies*, Montreal, Canada, 2009.

[103] Enrico Perla and Massimiliano Oldani. *A guide to kernel exploitation: attacking the core.* Syngress, 2010.

[104] Gilbert L Peterson and James S Okolica. Extracting Forensic Artifacts from Windows O/S Memory, 2011.

[105] Georgios Portokalidis and Herbert Bos. Eudaemon: Involuntary and on-demand emulation against zero-day exploits. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, Glasgow, Scotland, 2008.

[106] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on System Security*, Prague, Czech Republic, 2013.

[107] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, San Francisco, California, USA, 2012.

[108] Rasta ring 0 debugger. (http://rr0d.droids-corp.org/).

[109] Joanna Rutkowska. Subverting vista kernel for fun and profit. Black Hat USA.

[110] Joanna Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. *Invisible Things Lab.*, 2004.

[111] Joanna Rutkowska and Alexander Tereshkin. IsGameOver() anyone? Black Hat USA, 2007.

[112] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2001.

[113] Mathew J. Schwartz. Anonymous attacks north korea, denies targeting south, 2013. `http://www.informationweek.com/security/attacks/anonymous-attacks-north-korea-denies-tar/240157253`.

[114] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the Symposium on Foundations of Software Engineering*, Lisbon, Portugal, 2005.

[115] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[116] Amit Singh. *Mac OS X Internals: A Systems Approach.* Addison-Wesley Professional, 2006.

[117] Jim E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes.* Morgan Kaufmann, 2005.

[118] Mary Lou Soffa, Kristen R Walcott, and Jason Mars. Exploiting hardware advances for software testing and debugging (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, Honolulu, Hawaii, USA, 2011.

[119] SoftICE. (`http://en.wikipedia.org/wiki/SoftICE`).

[120] Passmark Software. PassMark Performance Test. `http://www.passmark.com/products/pt.htm`.

[121] R Stallman. The GNU debugger. Technical report, Technical report, Free Software Foundation, Inc, 675 Mass. Avenue, Cambridge, MA, 02139, USA, 1986.

[122] Elizabeth Stinson and John C. Mitchell. Characterizing bots' remote control behavior. In Christopher Kruegel, Richard Lippmann, and Andrew Clark, editors, *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*, San Diego, California, USA, 2007.

[123] Symantec. Symantec Report on the Underground Economy, 2008. http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_underground_economy_report_11-2008-14525717.en-us.pdf.

[124] Symantec. W32.Duqu: The Precursor to the Next Stuxnet, October 2011. http://www.symantec.com/connect/w32_duqu_precursor_next_stuxnet.

[125] Symantec. W32.Stuxnet Dossier, February 2011. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[126] Symantec. Flamer: Highly Sophisticated and Discreet Threat Targets the Middle East, May 2012. http://www.symantec.com/connect/blogs/flamer-highly-sophisticated-and-discreet-threat-targets-middle-east.

[127] Symantec. Internet security threat report, 2013. https://scm.symantec.com/resources/istr18_en.pdf.

[128] Syser kernel debugger. (http://www.sysersoft.com/).

[129] Peter Ször. Hunting for metamorphic. Technical report, Symantec, Inc., June 2003.

[130] Péter Ször. *The Art of Computer Virus Research and Defense.* Addison-Wesley, 2005.

[131] Ariel Tamches. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels.* PhD thesis, University of Wisconsin-Madison, 2001.

[132] The FreeBSD Project. FreeBSD. http://www.freebsd.org.

[133] The PaX Team. Address Space Layout Randomization. http://pax.grsecurity.net/docs/aslr.txt.

[134] Alan M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265, 1936.

[135] Amit Vasudevan and Ramesh Yerraballi. Stealth Breakpoints. In *In the proceedings of the 21st Annual Computer Security Applications Conference*, Tucson, Arizona, USA, 2005.

[136] VMware. http://www.vmware.com.

[137] VMware. Understanding full virtualization, paravirtualization, and hardware assist, 2007. `http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf`.

[138] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and Communications Security*, Washington, District of Columbia, USA, 2002.

[139] Kristen Walcott-Justice, Jason Mars, and Mary Lou Soffa. THeME: a system for testing by hardware monitoring events. In *Proceedings of the International Symposium on Software Testing and Analysis*, Minneapolis, Minnesota, USA, 2012.

[140] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, 2002.

[141] R. Wojtczuk. Subverting the Xen hypervisor. *Black Hat USA*, 2008, 2008.

[142] Jianyun Xu, Andrew H. Sung, Patrick Chavez, and Srinivas Mukkamala. Polymorphic malicious executable scanner by API sequence analysis. In *Proceedings of the 4th International Conference on Hybrid Intelligent Systems*, Kitakyushu, Japan, 2004.

[143] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.

[144] Lok Kwong Yan and Heng Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[145] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and Communications Security*, Alexandria, Virginia, USA, 2007.